Fortgeschrittene funktionale Programmierung
LVA 185.A05, VU 2.0, ECTS 3.0
SS 2016
(Stand: 23.06.2016)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen
Table of Contents
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 Summing Up

Part II: Programming Principles

- Chap. 2: Programming with Streams
  - 2.1 Streams
  - 2.2 Stream Diagrams
  - 2.3 Memoization
  - 2.4 Boosting Performance
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

- 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
Part III: Quality Assurance

- Chap. 5: Testing
  5.1 Defining Properties
  5.2 Testing against Abstract Models
  5.3 Testing against Algebraic Specifications
  5.4 Quantifying over Subsets
  5.5 Generating Test Data
  5.6 Monitoring, Reporting, and Coverage
  5.7 Implementation of QuickCheck
Table of Contents (4)

▶ Chap. 6: Verification

6.1 Equational Reasoning – Correctness by Construction
6.2 Basic Inductive Proof Principles
   6.2.1 Natural Induction
   6.2.2 Strong Induction
   6.2.3 Structural Induction
6.3 Inductive Proofs on Algebraic Data Types
   6.4.1 Induction and Recursion
   6.3.2 Inductive Proofs on Trees
   6.3.3 Inductive Proofs on Lists
   6.3.4 Inductive Proofs on Partial Lists
   6.3.5 Inductive Proofs on Streams
6.4 Approximation
6.5 Coinduction
6.6 Fixed Point Induction
6.7 Other Approaches, Verification Tools
Table of Contents (5)

Part IV: Advanced Language Concepts

- Chap. 7: Functional Arrays
- Chap. 8: Abstract Data Types
  8.1 Stacks
  8.2 Queues
  8.3 Priority Queues
  8.4 Tables
- Chap. 9: Monoids
- Chap. 10: Functors
  10.1 Motivation
  10.2 Constructor Class Functor
  10.3 Applicative Functors
  10.4 Kinds of Types and Type Constructors
# Table of Contents (6)

- **Chap. 11: Monads**
  - 11.1 Motivation
  - 11.2 Constructor Class Monad
  - 11.3 Predefined Monads
  - 11.4 Constructor Class MonadPlus
  - 11.5 Monadic Programming
  - 11.6 Monadic Input/Output
  - 11.7 A Fresh Look at the Haskell Class Hierarchy

- **Chap. 12: Arrows**
Part V: Applications

- Chap. 13: Parsing
  13.1 Combinator Parsing
  13.2 Monadic Parsing

- Chap. 14: Logical Programming Functionally

- Chap. 15: Pretty Printing

- Chap. 16: Functional Reactive Programming
  16.1 An Imperative Robot Language
  16.2 Robots on Wheels
  16.3 More on the Background of FRP
Table of Contents (8)

Part VI: Extensions and Prospectives

- Chap. 17: Extensions to Parallel and “Real World” Functional Programming
  17.1 Parallelism in Functional Languages
  17.2 Haskell for “Real World Programming”
- Chap. 18: Conclusions and Prospectives
- Bibliography
- Appendix
  - A Mathematical Foundations
    A.1 Sets and Relations
    A.2 Partially Ordered Sets
    A.3 Lattices
    A.4 Complete Partially Ordered Sets
    A.5 Fixed Point Theorems
    A.6 Cones and Ideals
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

Reconsidering a position statement by John Hughes that is based on an internal 1984 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 making 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 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

Next:

- Providing evidence for this claim
Functional programming owes its name to the facts 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)
Folk Knowledge: Soft Facts

...of 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)
Observation

...the commonly found previous 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.”
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

Open Issue:

- Why?
- Can it be concluded from the advantages of the “standard catalogue,” i.e., by dropping features?

Hardly.

This is not convincing. Overall, it reminds more to a medieval monk who denies himself the pleasures of life in the hope of getting virtuous.
Summing up: Lesson learnt

- The “standard catalogue” is not satisfying
  - 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
  - It does not provide 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
Towards a Positive Characterization

Structured vs. non-structured programming

...provides an analogue to compare with:

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

Note: This is essentially a negative "is-not"-characterization, too.
Towards a Positive Characterization (Cont’d)

Conceptually more important:

Structured programs are:

- designed modularly in contrast to non-structured programs
- Structured programming is more efficient/productive for this reason
  - Small modules are easier and faster 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”
The expressiveness of a language that 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

They offer conceptually new opportunities for modularization and re-use (beyond the more technical ones of lexical scoping, separate compilation, etc.), and make them more easily to achieve.

Modularization (smaller, simpler, more general) is the guideline, which should be followed by functional programmers in the course of programming.
In the following

- **Glueing functions together**
  - The clou: Higher-order functions

- **Glueing programs together**
  - The clou: Lazy evaluation
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)**

  \[
  [] \quad \text{means nil}
  \]

  \[
  [1] \quad \text{means cons 1 nil}
  \]

  \[
  [1,2,3] \quad \text{means cons 1 (cons 2 (cons 3 nil)))}
  \]

**Example:**

Adding the elements of a list

\[
\text{sum nil} = 0
\]

\[
\text{sum (cons num list)} = \text{num} + \text{sum list}
\]
Observation

Only the framed parts are specific to computing a sum:

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

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

...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

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

   This reveals the definition of \textit{reduce} almost immediately:

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

   Recall

   \[
   \begin{array}{|c|}
   \hline
   \text{sum \ nil} \\
   \hline
   \multicolumn{1}{|c|}{= \ | \ 0 \ |} \\
   \hline
   \end{array}
   \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ Quad
Immediate Benefit: Re-use

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

2. Computing the product of the elements of a list
   \[
   \text{product} = \text{reduce multiply 1}
   \]
   where \( \text{multiply x y} = x*y \)

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

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

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

- `cons` are replaced by \(f\)
- `nil` by \(a\)

Examples:

reduce `add` 0:

\[
\text{cons 1 (cons 2 (cons 3 nil))}
\]

\[
\rightarrow \quad \text{add 1 (add 2 (add 3 0))}
\]

\[
\rightarrow \quad 6
\]

reduce `multiply` 1:

\[
\text{cons 1 (cons 2 (cons 3 nil))}
\]

\[
\rightarrow \quad \text{multiply 1 (multiply 2 (multiply 3 1))}
\]

\[
\rightarrow \quad 6
\]
More Applications 1(5)

Observation: reduce cons nil copies a list of elements

This allows:

5. Concatenation of lists

\( \text{append a b} = \text{reduce cons b a} \)

Example:

\[
\text{append } [1,2] [3,4] \\
\rightarrow \rightarrow \text{reduce cons } [3,4] [1,2] \\
\rightarrow \rightarrow (\text{reduce cons } [3,4]) (\text{cons 1 (cons 2 nil)}) \\
\rightarrow \rightarrow \{ \text{replacing cons by cons and nil by } [3,4] \} \\
\text{cons 1 (cons 2 [3,4])} \\
\rightarrow \rightarrow \text{cons 1 (cons 2 (cons 3 (cons 4 nil)))} \\
\rightarrow \rightarrow [1,2,3,4]
\]
6. Doubling each element of a list

\[
doubleall = \text{reduce doubleandcons nil} \\
\text{where doubleandcons num list} \\
= \text{cons} (2*\text{num}) \text{ list}
\]
The function `doubleandcons` can be modularized further:

- **First step**
  
  ```
  doubleandcons = fandcons double
  where double n = 2*n
  fandcons f el list = cons (f el) list
  ```

- **Second step**
  
  ```
  fandcons f = cons . f
  where "." denotes the composition of functions:
  (f . g) h = f (g h)
  ```

**Note:** For checking correctness consider:

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

which yields as desired:

```
  fandcons f el list = cons (f el) list
```
Putting things together, we obtain:

6a. Doubling each element of a list
   
   doubleall = reduce (cons . double) nil

Another step of modularization using map leads us to:

6b. Doubling each element of a list
   
   doubleall = map double
   
   map f = reduce (cons . f ) nil

where map applies any function f to all the elements of a list.
After these preparative steps it is just as well possible:

7. Adding the elements of a matrix

    summatrix = sum . map sum

Homework: Think about how summatrix works.
Summing up

By decomposing (modularizing) and representing a simple function \( \text{sum} \) 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 \textit{without any further programming effort}! 
Generalization

...to more complex data structures:

**Trees:**

\[
\text{treeof } X ::= \text{node } X \ (\text{listof} \ (\text{treeof } X))
\]

**Example:**

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

```
Generalization (Cont’d)

Analogously to reduce on lists we introduce a functional redtree on trees:

\[
\text{redtree } f \text{ } g \text{ } a \text{ } (\text{node \ label \ subtrees}) \\
= f \text{ } \text{label} \text{ } (\text{redtree’ } f \text{ } g \text{ } a \text{ } \text{subtrees})
\]

where

\[
\text{redtree’ } f \text{ } g \text{ } a \text{ } (\text{cons \ subtree \ rest}) \\
= g \text{ } (\text{redtree } f \text{ } g \text{ } a \text{ } \text{subtree}) \text{ } (\text{redtree’ } f \text{ } g \text{ } a \text{ } \text{rest})
\]

\[
\text{redtree’ } f \text{ } g \text{ } a \text{ } \text{nil} \text{ } = \text{ } a
\]

Note: redtree takes 3 arguments \((f, g, a)\)

- The first one to replace \text{node} with
- The second one to replace \text{cons} with
- The third one to replace \text{nil} with
Applications 1(4)

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
Applications 2(4)

1. Adding the labels of the leaves of a tree

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

Example:
Using the tree introduced previously, we obtain:

   \[ \text{add 1} \]
   \[ (\text{add (add 2 0)} \]
   \[ \quad (\text{add (add 3} \]
   \[ \quad \quad (\text{add (add 4 0) 0})) \]
   \[ \quad \quad \quad 0)) \]
   \[ \rightarrow \rightarrow 10 \]
Applications 3(4)

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

labels = redtree cons append nil

Example:

cons 1
    (append (cons 2 nil)
        (append (cons 3
            (append (cons 4 nil) nil))
            nil))

->> [1,2,3,4]
Applications 4(4)

3. A function `maptree` on trees replicating the function `map` on lists

\[
\text{maptree } f = \text{redtree } (\text{node } . \ f) \text{ cons } \text{nil}
\]
Summing up

- The 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 almost without any further effort!
Summing up (Cont’d)

- **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).
Reminder to initial Thesis

- The expressiveness of a language that 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

  They offer conceptually new opportunities for modularization and re-use (beyond the more technical ones of lexical scoping, separate compilation, etc.), and make them more easily to achieve.

- Modularization (smaller, simpler, more general) is the guideline, which should be followed by functional programmers in the course of programming.
Reminder (Cont’d)

So far, we talked about:

- Higher-order functions as glue for glueing functions together

Next we will talk about:

- Lazy evaluation as glue for glueing programs together
Chapter 1.2

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 (such) programs, then also $g \circ f$ is a program. Applied to input as input, it yields the output $g(f\text{ input})$

- A possible implementation using conventional glue:
  - Communication via files
    - Possible problems
      - Temporary files can be too large
      - $f$ might not terminate
Functional Glue

Lazy evaluation allows a more elegant approach:

- **Decomposing** a problem into a
  - generator
  - selector

component, which are then **glued together**.

Intuition:

- The generator component “**runs as little as possible**” until it is terminated by the selector component.
Example 1: Computing Square Roots

Computing Square Roots (according to Newton-Raphson)

Given: \( N \)     Wanted: \( \text{squareRoot}(N) \)

Iteration formula:

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

Justification: If the approximations converge to some limit \( a \), we have:

\[
a = \frac{a + N/a}{2}
\]

\[
=> 2a = a + N/a
\]

\[
a = N/a
\]

\[
a*a = N
\]

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

I.e., \( a \) stores the value of the square root of \( N \).
For later comparison 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 monolithic, not decomposable.
The Functional Version 1(4)

Computing the next approximation from the previous one:

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

Introducing function \( f \) for the above computation, we are interested in computing the sequence of approximations:

\[ [a_0, f \ a_0, f(f \ a_0), f(f(f \ a_0)), ...] \]
The function \texttt{repeat} computes this (possibly infinite) sequence of approximations. It is the \texttt{generator} component in this example:

\textbf{Generator:}

\texttt{repeat} \texttt{f} \texttt{a} = \texttt{cons} \texttt{a} \texttt{(repeat} \texttt{f} \texttt{(f} \texttt{a}))

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

\texttt{repeat} \texttt{(next N)} \texttt{a0}
-\texttt{>} \texttt{[a0, f a0, f(f a0), f(f(f a0))],...}
The Functional Version 3(4)

**Note:** The evaluation of

```
repeat (next N) a0
```

does not terminate!

**Remedy:** Computing \texttt{squareroot} \, \texttt{N} up to a given tolerance \texttt{eps} \, \texttt{>} \, 0. Crucial: The \texttt{selector} component implemented by:

**Selector:**

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

**Final step:** Combining the components/modules:

```
\texttt{sqrt a0 eps N} = within eps (repeat (next N) a0)
```

\(\rightsquigarrow\) We are done!
The Functional Version 4(4)

Summing up:

- **repeat**: generator component:
  \[ [a_0, f \, a_0, f(f \, a_0), f(f(f \, a_0)), \ldots] \]
  ...potentially infinite, no limit on the length.

- **within**: selector component:
  \[ f^i \, a_0 \text{ with } \text{abs}(f^i \, a_0 - f^{i+1} \, a_0) \leq \varepsilon \]
  ...lazy evaluation ensures that the selector function
  is applied eventually ⇒ termination!

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

Next, we want to provide evidence that
generator
selector
can indeed be considered modules that can easily be re-used.

We are going to start with the re-use of the module generator.
Evidence of Generator-Modularity

Consider a new criterion for termination:

- Instead of awaiting the difference of successive approximations to approach zero ($\leq \epsilon$), await their ratio to approach one ($\leq 1+\epsilon$)

New Selector:

\[
\text{relative } \epsilon \text{ (cons } a \text{ (cons } b \text{ rest})) \\
= b, \quad \text{if } |a-b| \leq \epsilon \cdot |b| \\
= \text{relative } \epsilon \text{ (cons } b \text{ rest)}, \quad \text{otherwise}
\]

Final step: (Re-)combining the components/modules:

\[
\text{relative } \sqrt{a_0} \text{ } \epsilon \text{ } N \\
= \text{relative } \epsilon \text{ (repeat (next } N \text{) } a_0)
\]

\[\Rightarrow \text{ We are done!}\]
Note the Re-Use

...of the module generator in the previous example:

- The generator, i.e., the “module” computing the sequence of approximations has been re-used unchanged.

Next, we want to re-use the module selector.
Example 2: Numerical Integration

Numerical Integration

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

Wanted: The area under $f$ between $a$ and $b$

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

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

This is sufficiently precise, however, at most for very small intervals.
\[ \int_a^b f(x) \, dx = A + B = (f(a) + f(b)) \frac{(b-a)}{2} \]
Refinements 1(4)

Idea

- 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 \texttt{integrate} implements this strategy:

\textbf{Generator:}

\begin{verbatim}
integrate f a b = cons (easyintegrate f a b)
    \hspace{1cm} \text{map addpair (zip (integrate f a mid)
    \hspace{1cm} \hspace{1cm} (integrate f mid b)))}
    \hspace{1cm} \text{where mid = (a+b)/2}
\end{verbatim}

\textbf{Reminder:}

\begin{verbatim}
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
\end{verbatim}
Refinements 2(4)

- integrate is sound but inefficient (many redundant computations of \(f_a, f_b, \) and \(f_{mid}\))

The following version of \texttt{integrate} is free of this deficiency:

```haskell
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
```
Refinements 3(4)

Obviously, the evaluation of

\[ \text{integrate f a b} \]

does not terminate!

**Remedy:** Computing \( \text{integrate f a b} \) up to some limit \( \epsilon > 0 \).

**Two Selectors:**

- **Variant A:** within \( \epsilon \) (\( \text{integrate f a b} \))
- **Variant B:** relative \( \epsilon \) (\( \text{integrate f a b} \))
Refinements 4(4)

Summing up:

- **Generator component:**
  integrate
  ...potentially infinite, no limit on the length.

- **Selector component:**
  within, relative
  ...lazy evaluation ensures that the selector function is applied eventually ⇒ termination!
Note the Re-Use

...of the module selector in the previous example:

- The selector, i.e., the “module” picking the solution from the stream of approximate solutions has been re-used unchanged.

Again, lazy evaluation is the key to synchronize the generator and selector module!
Example 3: Numerical Differentiation

Numerical Differentiation

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

Wanted: The slope of \( f \) at point \( x \)

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

\[
easydiff f x h = \frac{f(x+h) - f(x)}{h}
\]

This is sufficiently precise, however, at most for very small values of \( h \).
Refinements

Generate a sequence of approximations getting successively “better”:

**Generator:**

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

Select a sufficiently precise approximation:

**Selector:**

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

Implementing the selector: Homework
The Generator/Selector Principle at a Glance

**Generator**

iterate \( f \times \)

\[ x, f \times, f(f \times), \ldots \]

**Selector/Filter**

select \( p \)

\[ x, y, z, \ldots \]

\[ q \mid q \leftarrow [x, y, z, \ldots], \]

\[ \text{select } p \ q == True \]

**Combining Generator and Selector/Filter**

iterate \( f \times \)

select \( p \)

\[ x, f \times, f(f \times), \ldots \]

\[ q \mid q \leftarrow [x, f \times, f(f \times), \ldots], \]

\[ \text{select } p \ q == True \]
The Generator/Transformer Princ. at a Glance

**Generator**

iterate \( f \ x \)

\[ x, f \ x, f(f \ x), \ldots \]

**Transformer**

map \( g \)

\[ x, y, z, \ldots \]

\[ g \ x, g \ y, g \ z, \ldots \]

**Combining Generator and Transformer**

iterate \( f \ x \)

map \( g \)

\[ x, f \ x, f(f \ x), \ldots \]

\[ g \ x, g(f \ x), g(f(f \ x)), \ldots \]
The composition pattern, which in fact is common to all three examples becomes again obvious. It consists of a

- **generator** (usually looping!) and
- **selector** (ensuring termination thanks to lazy evaluation!)
Summary of Findings (2)

Thesis

- 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!
Summary of Findings (3)

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 makes functional programs to be written so concisely and elegantly (not the absence of assignments, etc.)
Summary of Findings (4)

Guidelines

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

are the tools for achieving this!
Chapter 1.4
Summing Up
Summing Up: Lazy or Eager Evaluation

The final conclusion of John Hughes:

- In view of the previous arguments:
  - 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.
Outlook

John Hughes identifies

- higher-order functions
- lazy evaluation

as of vital importance for the power of the functional programming style.

In Chapter 2 and in Chapter 3 we will discuss the power they provide the programmer with in more detail:

- **Stream programming**: thanks to lazy evaluation.
- **Algorithm patterns**: thanks to higher-order functions.
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 = Infinite Lists

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
  ▶ Proving properties of programs with streams
Chapter 2.1
Streams
Streams

Jargon

Stream ...synonymous to infinite list and lazy list.

Streams

- (combined with lazy evaluation) allow to solve many problems elegantly, concisely, and efficiently
- are a source of hassle if applied inappropriately

More on this in this chapter.
Streams

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

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

**Convention**

For pragmatic reasons, however, we will model streams as ordinary lists waiving the usage of the empty list `[]`.

This is motivated by:

- **Convenience/adequacy** because many pre-defined (polymorphic) functions on lists can be reused this way, which otherwise would have to be defined from scratch on the new data type `Stream`
First Examples of Streams

- **Built-in streams in Haskell**
  
  
  \[
  [2..] \rightarrow [2,3,4,5,6,7,\ldots] \\
  [3,5..] \rightarrow [3,5,7,9,11,\ldots]
  \]

- **User-defined streams in Haskell**
  
  The infinite lists of “twos”

  \(2,2,2,\ldots\)

  In Haskell this can be realized:

  - using list comprehension: \([2,2..]\)
  - (co-) recursively: \(\text{twos} = 2 : \text{twos}\)

  Illustration

  \[
  \text{twos} \rightarrow 2 : \text{twos} \\
  \rightarrow 2 : 2 : \text{twos} \\
  \rightarrow 2 : 2 : 2 : \text{twos} \\
  \rightarrow \ldots
  \]

  \text{twos} represents an infinite list; synonymously, 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” look like recursive definitions.

- However, they lack a base case.

- Definitions of the above form are called corecursive

- Corecursive definitions always yield infinite objects.
More corecursively defined Streams

- The stream of natural numbers \( \text{nats} \)
  \[
  \text{nats} = 0 : \text{map (+1)} \text{nats}
  \]

- The stream of even natural numbers \( \text{evens} \)
  \[
  \text{evens} = 0 : \text{map (+2)} \text{evens}
  \]

- The stream of odd natural numbers \( \text{odds} \)
  \[
  \text{odds} = 1 : \text{map (+2)} \text{odds}
  \]
More Streams

- The stream of natural numbers
  \[
  \text{theNats} = 0 : \text{zipWith (+) ones theNats}
  \]

- The stream of powers of an integer
  \[
  \text{powers} :: \text{Int} \rightarrow [\text{Int}]
  \]
  \[
  \text{powers n} = [n^{x} \mid x \leftarrow [0..]]
  \]

- The prelude function \text{iterate}
  \[
  \text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]
  \]
  \[
  \text{iterate f x} = x : \text{iterate f (f x)}
  \]
  The function \text{iterate} generates the stream
  \[
  [x, f x, (f . f) x, (f . f . f) x, \ldots]
  \]

  Application: \text{powers} can be defined in terms of \text{iterate}
  \[
  \text{powers n} = \text{iterate} ((\ast n) 1)
  \]
More Applications of iterate

ones    = iterate id 1

twos    = iterate id 2

threes  = iterate id 3

nats    = iterate (+1) 0

evens   = iterate (+2) 0

odds    = iterate (+2) 1

powers  = iterate (*n) 1
Functions on Streams

\[ \text{head} :: [a] \to a \]
\[ \text{head} \ (x:_) = x \]

Application

\[ \text{head} \ \text{twos} \rightarrow_\rightarrow \text{head} \ (2 : \ \text{twos}) \rightarrow_\rightarrow 2 \]

Note: Normal-order reduction (resp. its efficient implementation variant lazy evaluation) ensures termination in this example. It excludes the infinite sequence of reductions:

\[ \text{head} \ \text{twos} \]
\[ \rightarrow_\rightarrow \text{head} \ (2 : \ \text{twos}) \]
\[ \rightarrow_\rightarrow \text{head} \ (2 : 2 : \ \text{twos}) \]
\[ \rightarrow_\rightarrow \text{head} \ (2 : 2 : 2 : \ \text{twos}) \]
\[ \rightarrow_\rightarrow \ldots \]
Reminder

“...whenever there is a terminating reduction sequence of an expression, then normal-order reduction terminates.”

(Church/Rosser-Theorem)

- Normal-order reduction corresponds to leftmost-outermost evaluation

Recall: Let

\[ \text{ignore} :: a \rightarrow b \rightarrow b \]
\[ \text{ignore} \ a \ b = b \]

Then, both in

- \text{ignore twos 42}
- twos 'ignore' 42

the leftmost-outermost operator is given by the call \text{ignore}.
Functions on Streams (Cont’d)

\[
\text{addFirstTwo} :: [\text{Integer}] \to \text{Integer}
\text{addFirstTwo} (x:y:zs) = x+y
\]

Application

\[
\text{addFirstTwo} \; \text{twos} \Leftrightarrow \text{addFirstTwo} \; (2:\text{twos})
\to \text{addFirstTwo} \; (2:2:\text{twos})
\to 2+2
\to 4
\]
Functions yielding Streams

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

  \[ \text{from} :: \text{Int} \rightarrow \text{[Int]} \]
  \[ \text{from } n = n : \text{from} \ (n+1) \]

  \[ \text{fromStep} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{[Int]} \]
  \[ \text{fromStep } n \ m = n : \text{fromStep} \ (n+m) \ m \]

**Applications**

\[ \text{from } 42 \rightarrow> [42, 43, 44, \ldots] \]

\[ \text{fromStep } 3 \ 2 \rightarrow> 3 : \text{fromStep} \ 5 \ 2 \]
\[ \rightarrow> 3 : 5 : \text{fromStep} \ 7 \ 2 \]
\[ \rightarrow> 3 : 5 : 7 : \text{fromStep} \ 9 \ 2 \]
\[ \rightarrow> \ldots \]
Primes: The Sieve of Eratosthenes 1(3)

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...
Primes: The Sieve of Eratosthenes 2(3)

The stream of primes:

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

sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, mod y x > 0]
```
Illustration: By stepwise evaluation

```
primes
    ->> sieve [2..]
    ->> 2 : sieve [ y | y <- [3..], mod y 2 > 0]
    ->> 2 : sieve (3 : [ y | y <- [4..], mod y 2 > 0]
    ->> 2 : 3 : sieve [ z | z <- [ y | y <- [4..],
                          mod y 2 > 0 ],
                       mod z 3 > 0]
    ->> ...  
    ->> 2 : 3 : sieve [ z | z <- [5, 7, 9..],
                       mod z 3 > 0]
    ->> ...  
    ->> 2 : 3 : sieve [5, 7, 11,...
    ->> ...  
```
Pitfalls in Applications

Implementing a prime number test (naively):

Let

\[
\text{member} :: [a] \rightarrow a \rightarrow \text{Bool}
\]
\[
\text{member} \ [\] \ y = \text{False}
\]
\[
\text{member} \ (x:xs) \ y = (x==y) || \text{member} \ xs \ y
\]

Then

\[
\text{member} \ \text{primes} \ 7 \ \ldots \text{yields } \text{"True" (as expected!)}
\]

But

\[
\text{member} \ \text{primes} \ 6 \ \ldots \text{does not terminate!}
\]

Homework: Why fails the above implementation? How can \text{primes} be embedded into a calling context allowing us to decide if some argument is prime or not?
Random Numbers 1(2)

Generating a sequence of (pseudo-) random numbers:

\[
\text{nextRandNum} :: \text{Int} \rightarrow \text{Int} \\
\text{nextRandNum} \ n = (\text{multiplier} \times n + \text{increment}) \mod \text{modulus}
\]

\[
\text{randomSequence} :: \text{Int} \rightarrow \left[\text{Int}\right] \\
\text{randomSequence} = \text{iterate} \ \text{nextRandNum}
\]

Choosing

\[
\begin{align*}
\text{seed} & = 17489 \\
\text{increment} & = 13849 \\
\text{multiplier} & = 25173 \\
\text{modulus} & = 65536
\end{align*}
\]

we obtain the following sequence of (pseudo-) random numbers

\[
[17489, 59134, 9327, 52468, 43805, 8378, \ldots]
\]
ranging from 0 to 65536, where all numbers of this interval occur with the same frequency.
Random Numbers 2(2)

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

This can be achieved by scaling the sequence.

\[
\text{scale} :: \text{Float} \to \text{Float} \to \text{[Int]} \to \text{[Float]}
\]
\[
\text{scale } \text{p } \text{q } \text{randSeq} = \text{map } (\text{f } \text{p } \text{q}) \text{randSeq}
\]
where \( \text{f} :: \text{Float} \to \text{Float} \to \text{Int} \to \text{Float} \)
\[
\text{f } \text{p } \text{q } \text{n} = \text{p } + \left( (\text{n } \times (\text{q}-\text{p})) / (\text{modulus}-1) \right)
\]

**Application**

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

- Other combinations of **generators**, **filters**, and **selectors**
The Fibonacci Numbers 1(5)

The sequence of **Fibonacci Numbers**

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

is defined in terms of the function

\[\text{fib} : \mathbb{N} \rightarrow \mathbb{N}\]

\[\text{fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise}
\end{cases}\]
The Fibonacci Numbers 2(5)

We have already learned that a **naive implementation** like

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

...that **directly** exploits the recursive pattern of the underlying mathematical function is

- inacceptably inefficient and slow!
The Fibonacci Numbers 3(5)

Illustration: By stepwise evaluation

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

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

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

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

fib 4 $\Rightarrow$ fib 3 + fib 2
$\Rightarrow$ (fib 2 + fib 1) + (fib 1 + fib 0)
$\Rightarrow$ ((fib 1 + fib 0) + 1) + (1 + 0)
$\Rightarrow$ ((1 + 0) + 1) + (1 + 0)
$\Rightarrow$ 3 $\quad$ 9 calls of fib

fib 5 $\Rightarrow$ fib 4 + fib 3
$\Rightarrow$ (fib 3 + fib 2) + (fib 2 + fib 1)
$\Rightarrow$ ((fib 2 + fib 1) + (fib 1 + fib 0)) + ((fib 1 + fib 0) + 1)
$\Rightarrow$ (((fib 1 + fib 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
$\Rightarrow$ (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
$\Rightarrow$ 5 $\quad$ 15 calls of fib
The Fibonacci Numbers 5(5)

\[
\begin{align*}
\text{fib 8} & \rightarrow \text{fib 7} + \text{fib 6} \\
& \rightarrow (\text{fib 6} + \text{fib 5}) + (\text{fib 5} + \text{fib 4}) \\
& \rightarrow ((\text{fib 5} + \text{fib 4}) + (\text{fib 4} + \text{fib 3})) \\
& \quad + ((\text{fib 4} + \text{fib 3}) + (\text{fib 3} + \text{fib 2})) \\
& \rightarrow (((\text{fib 4} + \text{fib 3}) + (\text{fib 3} + \text{fib 2})) \\
& \quad + (\text{fib 3} + \text{fib 2}) + (\text{fib 2} + \text{fib 1}))) \\
& \quad + (((\text{fib 3} + \text{fib 2}) + (\text{fib 2} + \text{fib 1})) \\
& \quad + ((\text{fib 2} + \text{fib 1}) + (\text{fib 1} + \text{fib 0}))) \\
& \rightarrow \ldots \\
& \rightarrow 21 \quad -- \quad 60 \text{ calls of fib}
\end{align*}
\]

...tree-like recursion (with \textbf{exponential growth}!)
Reminder: Complexity 1(3)


Reminder: $\mathcal{O}$ 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 $\mathcal{O}(f)$ denotes the set of all functions which “grow slower” than $f$:

$$\mathcal{O}(f) = \{ h \mid h(n) \leq c \cdot f(n) \text{ for some positive constant } c \text{ and all } n \geq N_0 \}$$
### Reminder: Complexity 2(3)

**Examples of typical cost functions:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Costs</th>
<th>Intuition: <em>input a thousandfold as large means:</em></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>“$n \log n$”</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>...billiardfold 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>
Reminder: Complexity 3(3)

...and the impact of growing inputs in practice in hard numbers:

<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>
Remedy

- Streams can (often) help!
Fibonacci Numbers Efficiently 1(2)

Idea

0 1 1 2 3 5 8 13... Sequence of Fib. Numbers
1 1 2 3 5 8 13 21... Remainder of the S. of F. N.

1 2 3 5 8 13 21 34... Remain. of the rem. of the sequ. of Fibonacci Numbers

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

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

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

...reminds to Münchhausen’s famous trick of “sich am eigenen Schopfe aus dem Sumpf zu ziehen!”
Fibonacci Numbers Efficiently 2(2)

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

take 10 fibs ->> [0,1,1,2,3,5,8,13,21,34]

where

\[
\begin{align*}
\text{take} & : \text{Integer} \rightarrow [a] \rightarrow [a] \\
\text{take} 0 \_ & = [] \\
\text{take} \_ [] & = [] \\
\text{take} n (x:xs) \mid n > 0 & = x : \text{take} (n-1) xs \\
\text{take} \_ \_ & = \text{error} \ "\text{PreludeList.take: negative argument}"
\end{align*}
\]
Summing up

We get a conceptually new implementation of the Fibonacci function using corecursive streams:

\[
\text{fib} :: \text{Int} \rightarrow \text{Integer} \\
\text{fib} \ n = \text{last} \ (\text{take} \ n \ \text{fibs})
\]

Even shorter:

\[
\text{fib} :: \text{Int} \rightarrow \text{Integer} \\
\text{fib} \ n = \text{fibs!!} (n-1)
\]

Remark:
Note the application of the
- Generator/Selector Principle
in this example.
Naive Evaluation (no sharing)

...stepwise evaluation (with add instead of zipWith (+)):

fibs

->> Replace the call of fibs by the body of fibs
   0 : 1 : add fibs (tail fibs)
->> Replace both calls of fibs by the body of fibs
   0 : 1 : add (0 : 1 : add fibs (tail fibs))
   (tail (0 : 1 : add fibs (tail fibs)))
->> Application of tail
   0 : 1 : add (0 : 1 : add fibs (tail fibs))
   (1 : add fibs (tail fibs))
->> ...

- Observation: The computational effort remains exponential this (naive) way!
- Clou: Lazy evaluation – common subexpressions will not be computed multiple times (in the example this holds for tail and fibs)!
The Benefit of Lazy Evaluation (sharing) 1(3)

fibs $\rightarrow\rightarrow$ 0 : 1 : add fibs (tail fibs)

$\rightarrow\rightarrow$ Introduc. abbrev. allows sharing of results

0 : tf  (tf reminds to "tail of fibs")
where tf = 1 : add fibs (tail fibs)

$\rightarrow\rightarrow$ 0 : tf
where tf = 1 : add fibs tf

$\rightarrow\rightarrow$ Introducing abbreviations allows sharing

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

$\rightarrow\rightarrow$ Unfolding of add

0 : tf
where tf = 1 : tf2
where tf2 = 1 : add tf tf2
The Benefit of Lazy Evaluation (sharing) 2(3)

- Repeating the above steps
  0 : tf
  where tf = 1 : tf2
  where tf2 = 1 : tf3 (tf3 reminds to "tail of tail of tail of fibs")
  where tf3 = add tf tf2

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

- tf is only used at one place and can thus be eliminated
  0 : 1 : tf2
  where tf2 = 1 : tf3
  where tf3 = 2 : add tf2 tf3
Finally, we obtain successsively longer prefixes of the stream of Fibonacci numbers

0 : 1 : tf2

where tf2 = 1 : tf3

where tf3 = 2 : tf4

where tf4 = add tf2 tf3

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

0 : 1 : tf2

where tf2 = 1 : tf3

where tf3 = 2 : tf4

where tf4 = 3 : add tf3 tf4

0 : 1 : 1 : tf3

where tf3 = 2 : tf4

where tf4 = 3 : add tf3 tf4
Pitfall

In practice, the ability of dividing/recognizing common structures is limited.

This is demonstrated by the below variant of the Fibonacci function that artificially lifts `fibs` to a functional level:

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

This function again exposes

- exponential run-time and storage behaviour!

Crucial:

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

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

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

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

In a special case like here, this is possible, but there is no general means for detecting such equalities!
Chapter 2.2
Stream Diagrams
Stream Diagrams

Problems on streams can often be considered and visualized as

- processes.

In the following, we consider two examples:

- The stream of Fibonacci numbers
- The communication stream of a client/server application
Fibonacci Numbers

... as a stream diagram:

\[ \text{fibs} = 0, 1, 1, 2, 3, 5, 8, \ldots \]

\[ 1, 1, 2, 3, 5, 8, \ldots \]

\[ 0 \]

\[ 1 \]

\[ \text{add} \]
The Client/Server Application

Interaction of a server and a client (e.g. Web server/Web browser):

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

reqs = client resps
resps = server reqs

Implementation

type Request = Integer
type Response = Integer

client ys = 1 : ys \ (issues 1 as first request and then each integer it receives from the server)

server xs = map (+1) xs \ (adds 1 to each request it receives)
The Client/Server Application (Cont’d)

**Illustration:** By stepwise evaluation

reqs \(\rightarrow\) client resps
\(\rightarrow\) 1 : resps
\(\rightarrow\) 1 : server reqs

\(\rightarrow\) **Introducing abbreviations**

1 : tr
where tr = server reqs

\(\rightarrow\) 1 : tr
where tr = 2 : server tr

\(\rightarrow\) 1 : tr
where tr = 2 : tr2
where tr2 = server tr
The Client/Server Application (Cont’d)

\[
\begin{align*}
\text{->} & \quad 1 : \ tr \\
& \text{where } tr = 2 : \ tr2 \\
& \quad \text{where } tr2 = 3 : \text{ server } tr2 \\
\text{->} & \quad 1 : 2 : \ tr2 \\
& \text{where } tr2 = 3 : \text{ server } tr2 \\
\text{->} & \quad \ldots
\end{align*}
\]

In particular, we obtain:

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

... as a stream diagram:

```
1

server

resps = 2,3,4,5,...

client

(:)

1

reqs = 1,2,3,4,5,...

server

(+1)
```
Pitfall

Suppose, the client wants to check the first response:

\[
\text{client } (y:y') = \begin{cases} 
1 : (y:y') & \text{if ok } y \\
\text{error "Faulty Server"} & \text{else}
\end{cases}
\]

where

\[
\text{ok } y = \text{True} \quad (\text{Obviously a trivial predicate})
\]

The evaluation of:

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

...does not terminate!

The problem: Livelock! Neither the client nor the server can be unfolded! Pattern matching is too “eager.”
Remedy: Lazy Patterns 1(3)

Ad-hoc Remedy:

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

- Replacing of pattern matching by an explicit usage of the selector function `head`.
- Moving the conditional inside of the list.
Remedy: Lazy Patterns 2(3)

Systematic remedy: Lazy patterns

- **Syntax:** Preceding tilde (∼)
- **Effect:** Like using an explicit selector function; pattern-matching is deferred

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

**Note:** Even when using a lazy pattern the conditional must still be moved. **But:** The explicit usage of the selector function is avoided!

In practice, this can be very many selector functions that are saved this way making the programs “more” declarative and readable.
Remedy: Lazy Patterns 3(3)

Illustration: By stepwise evaluation

\[
\begin{align*}
\text{reqs} & \rightarrow\rightarrow \text{client resps} \\
& \rightarrow\rightarrow 1 \, : \, \text{if ok y then y : ys} \\
& \quad \text{else error "Faulty Server"} \\
& \quad \text{where y:ys = resps} \\
& \rightarrow\rightarrow 1 \, : \, (y:ys) \\
& \quad \text{where y:ys = resps} \\
& \rightarrow\rightarrow 1 \, : \, \text{resps}
\end{align*}
\]
Chapter 2.3
Memoization
Motivation

Memoization

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

that benefits from

- stream programming.
Memoization

The concept of


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 **memo function** is used to replace a costly to compute function by a (memo) table look-up. Intuitively, the original function is augmented by a cache storing argument/result pairs.
Memo Functions, Memo Tables

A memo function is

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

A memo table allows

- to replace recomputation by table look-up.

Correctness of the overall approach:

- Referential transparency of functional programming languages (in particular, absence of side effects!).
Memo Functions, Memo Tables (Cont’d)

A memo function `memo` associated with a function `f`

\[
\text{memo} :: (a \rightarrow b) \rightarrow (a \rightarrow b)
\]

has to be defined such that the following equality holds:

\[
\text{memo } f \ x = f \ x
\]
A Concrete Approach with Memo Lists

Memo List:
The (generic) memo function/table

\[ \text{flist} = [ f \; x \mid x \leftarrow [0..] ] \]

...where \( f \) is a function on integers.

Application:
Each call of \( f \) is replaced by a look-up in \( \text{flist} \).
Example 1: Computing Fibonacci Numbers

Computing Fibonacci numbers with memoization:

```hs
fiblist = [ fibm x | x <- [0..] ]
fibm 0 = 0
fibm 1 = 1
fibm n = fiblist !! (n-1) + fiblist !! (n-2)
```

Compare this with the naive implementation of `fib`:

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

Note:

```hs
fibm n = fib n
```
Example 2: Computing Powers

Computing powers ($2^0$, $2^1$, $2^2$, $2^3$, ...) with memoization:

```haskell
powerlist = [ powerm x | x <- [0..] ]
powerm 0  = 1
powerm i  = powerlist !! (i-1) + powerlist !! (i-1)
```

Compare this with the naive implementation of power:

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

Observation:

- Looking-up the result of the second call instead of recomputing it requires only $1 + n$ calls of `power` instead of $1 + 2^n$

$\rightarrow$ Significant performance gain!
The function \texttt{memo} :: (a \to b) \to (a \to b):

\begin{itemize}
  \item is essentially the identity on functions but
  \item \texttt{memo} keeps track on the arguments, it has been applied to and the corresponding results
\end{itemize}

\textbf{Motto:} look-up a result that has been computed previously instead of recomputing it!

\textbf{Memo functions}

\begin{itemize}
  \item are not part of the Haskell standard, but there are nonstandard libraries
\end{itemize}
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
mfibsFn :: () -> [Integer]
mfibsFn x
    = let mfibs = memo1 mfibsFn in
        0 : 1 : zipWith (+) (mfibs ()) (tail (mfibs ()))
```
Summing up (Cont’d)

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

- Chapter 19, Memoization

- Chapter 12.3, Memoization
Summing up (Cont’d)

- (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.

Two techniques that can (often) help achieving this are:

▸ Stream programming
▸ Memoization
Avoiding Recomputations using Stream Prog.

- Computing Fibonacci numbers using stream prog.:
  
  ```haskell
  fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
  
take 10 fibs -&gt; [0,1,1,2,3,5,8,13,21,34]
fibs!!5 -&gt; 5
  ```

- Computing powers using stream programming:
  
  ```haskell
  powers :: [Integer]
powers = 1 : 2 :
    zipWith (+) (tail powers) (tail powers)
  
take 9 powers -&gt; [1,2,4,8,16,32,64,128,256]
powers!!5 -&gt; 32
  ```

- ...
Avoiding Recomputations using Memoization

- **Computing Fibonacci numbers with memoization:**
  
  \[
  \text{fiblist} = \left[ \text{fibm} \ x \mid x \leftarrow [0..] \right]
  \]
  
  \[
  \text{fibm} \ 0 = 0
  \]
  
  \[
  \text{fibm} \ 1 = 1
  \]
  
  \[
  \text{fibm} \ n = \text{fiblist}!!(n-1) + \text{fiblist}!!(n-2)
  \]
  
  \[
  \text{take 10 fiblist} -\rightarrow \ [0,1,1,2,3,5,8,13,21,34]
  \]
  
  \[
  \text{fiblist}!!5 -\rightarrow 5
  \]

- **Computing powers with memoization:**

  \[
  \text{powerlist} = \left[ \text{powerm} \ x \mid x \leftarrow [0..] \right]
  \]

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

  \[
  \text{powerm} \ i = \text{powerlist}!!(i-1) + \text{powerlist}!!(i-1)
  \]

  \[
  \text{take 9 powerlist} -\rightarrow \ [1,2,4,8,16,32,64,128,256]
  \]

  \[
  \text{powerlist}!!5 -\rightarrow 32
  \]

- ...
Summing up

Stream programming and memoization are 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 solution to identical subproblems, e.g. tree-recursive processes.

Homework: Compare the performance of the straightforward implementations of \texttt{fib} and \texttt{power} with their “boosted” versions using stream programming and memoization.
Silver Bullets exist Sometimes

Though not in general, it is worth noting that sometimes there is a silver bullet solving a problem:

The computation of the Fibonacci numbers is again a striking example.

We can prove (cf. Chapter 6) the following theorem that allows a recursion-free direct computation of the Fibonacci numbers, i.e.,

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

Theorem

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

The usage of streams (and lazy evaluation) is advocated by:

- **Higher abstraction**: limitations to finite lists are 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: 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 (6)


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.
Motivation (Cont’d)

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

In detail:

- **Top-down**: starting from the initial problem, the algorithm works down to the solution by considering alternatives.
  - Divide-and-conquer
  - 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:
Let $P$ be a problem specification.

Solving $P$ – The Idea:

- If the problem is simple/small (enough), solve it directly or by means of some basic algorithm.
- Otherwise, divide the problem into smaller subproblems applying the division strategy recursively until all subproblems are simple enough to be directly solved.
- Combine all the solutions of the subproblems into a single solution of the initial problem.
Illustrating the Divide-and-Conquer Principle

Successive stages in a divide-and-conquer algorithm:

initial problem → subproblems → indivisible problems → solutions of subproblems → solution of the initial problem

divide phase

solve phase

combine phase

Implementing Divide-and-Conquer as HOF (1)

The Initial Setting:

- A problem with
  - problem instances of kind $p$

and solutions with

- solution instances of kind $s$

Objective:

- A higher-order function (HOF) `divideAndConquer`
  - solving suitably parameterized problem instances of kind $p$ utilizing the “divide and conquer” principle.
Implementing Divide-and-Conquer as HOF (2)

The ingredients of `divideAndConquer`:

- `indiv :: p -> Bool`: The function `indiv` yields `True`, if the problem instance can/need not be divided further (e.g., it can directly be solved by some `basic` algorithm).
- `solve :: p -> s`: The function `solve` yields the solution instance of a problem instance that cannot be divided further.
- `divide :: p -> [p]`: The function `divide` divides a problem instance into a list of subproblem instances.
- `combine :: p -> [s] -> s`: Given the original problem instance and the list of the solutions of the subproblem instances derived from, the function `combine` yields the solution of the original problem instance.
Implementing Divide-and-Conquer as HOF (3)

The HOF-Implementation:

\[
\text{divideAndConquer ::} \\
(p \to \text{Bool}) \to (p \to \text{s}) \to (p \to [p]) \to \\
(p \to [s] \to \text{s}) \to p \to \text{s}
\]

\[
\text{divideAndConquer indiv solve divide combine initPb} = \text{dAC initPb} \\
\text{where} \\
\text{dAC pb} \\
| \text{indiv pb} = \text{solve pb} \\
| \text{otherwise} = \text{combine pb (map dAC (divide pb))}
\]
Typical Applications of Divide-and-Conquer

Typical Applications:

- Application areas such as
  - Numerical analysis
  - Cryptography
  - Image processing
  - ...
- Quicksort
- Mergesort
- Binomial coefficients
- ...
Divide-and-Conquer for Quicksort

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

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

Consider:

```haskell
fib :: Integer -> Integer
fib n
  = divideAndConquer indiv solve divide combine n
  where
    indiv n       = (n == 0) || (n == 1)
    solve n
      | n == 0  = 0
      | n == 1  = 1
      | otherwise = error "solve: problem divisible"
    divide n      = [n-2,n-1]
    combine _ [l1,l2] = l1 + l2
```

...shows **exponential** runtime behaviour due to recomputations!
Illustration

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

```
fib 4
fib 3
fib 2
fib 2
fib 1
fib 1
fib 1
fib 0
fib 1
fib 0
```

Chapter 3.2
Backtracking Search
Backtracking Search

Given:
Let $P$ be a problem specification.

Solving $P$ – The Idea

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

Main Problem Characteristics for Applicability

- 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 Backtracking Search

General stages in a backtracking algorithm:

Fethi Rabhi, Guy Lapalme. 
Addison-Wesley, 1999, page 162.
Illustrating Backtracking Search (Cont’d)

Intuitively

- When exploring the graph, each visited path can lead to the goal node with an equal chance.
- Sometimes, however, there can be a situation, in which it is 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)

The Initial Setting:

- A problem with
  - problem instances of kind $p$

  and solutions with
  - solution instances of kind $s$

Objective:

- A higher-order function (HOF) `searchDfs`
  - solving suitably parameterized problem instances of kind $p$ utilizing the “backtracking” principle.
Implementing Backtracking Search as HOF (2)

Often:

- The search space is large.

Hence, the graph representing the search space

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

This requires:

- An appropriate type `node` that represents node information
- a successor function `succ` of type `node -> [node]` that generates the list of successors of a node.
Implementing Backtracking Search as HOF (2)

Assumptions:

- an acyclic implicit graph
- all solutions shall be computed (not only the first one)

Note: The HOF can be adjusted to terminate after finding the first solution.

The ingredients of `searchDfs`:

- `node`: A type representing node information.
- `succ :: node -> [nodes]`: The function `succ` yields the list of successors of a node.
- `goal :: node -> Bool`: The function `goal` determines if a node is a solution.
Implementing Backtracking Search as HOF (3)

The HOF-Implementation:

```
searchDfs ::
  (Eq node) => (node -> [node]) -> (node -> Bool)
  -> node -> [node]

searchDfs succ goal x
  = (search' (push x emptyStack) )
  where
    search' s
      | stackEmpty s = []
      | goal (top s) = top s : search' (pop s)
      | otherwise
        = let x = top s
            in search' (foldr push (pop s) (succ x))
```
The Abstract Data Type Stack (1)

The user-visible interface specification of the Abstract Data Type (ADT) Stack:

module Stack (Stack,push,pop,top,
    emptyStack,stackEmpty) where

push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
emptyStack :: Stack a
stackEmpty :: Stack a -> Bool
The Abstract Data Type Stack (2)

A user-invisible implementation of Stack as an algebraic data type (using data):

```haskell
data Stack a = EmptyStk
              | Stk a (Stack a)

push x s = Stk x s

pop EmptyStk = error "pop from an empty stack"
pop (Stk _ s) = s

top EmptyStk = error "top from an empty stack"
top (Stk x _) = x

emptyStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _ = False
```
The Abstract Data Type Stack (3)

A user-invisible implementation of Stack as an algebraic data type (using newtype):

newtype Stack a = Stk [a]

push x (Stk xs) = Stk (x:xs)

pop (Stk []) = error "pop from an empty stack"
pop (Stk (_:xs)) = Stk xs

top (Stk []) = error "top from an empty stack"
top (Stk (x:_)) = x

emptyStack = Stk []

stackEmpty (Stk []) = True
stackEmpty (Stk _) = False
Typical Applications of Backtracking Search

Typical Applications:

- Application areas such as
  - game strategies
  - ...
- The eight-tile problem (8TP)
- The $n$-queens problem
- Towers of Hanoi
- The knapsack problem
The Eight-Tile Problem

Fethi Rabhi, Guy Lapalme.  
A Backtracking Search 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 for 8TP (2)

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

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

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

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

...the list of configurations reachable in one move is obtained by placing the space at position $i$ and indicating that tile $i$ is now where the space was.
A Backtracking Search 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 -> [Boards]
succ8Tile (BDS(n@(b:bs)))
  = filter (notIn bs) [BDS(b’:n) | b’ <- allMoves b]
  where
    notIn bs (BDS(b:_))
      = not (elem (elems b) (map elems bs))
```

...computes all successors that have not been encountered before; the `notIn`-test ensures that only nodes are considered that have not been encountered before.
A Backtracking Search for 8TP (4)

The goal function:

\[
\text{goal8Tile} :: \text{Boards} \rightarrow \text{Bool} \\
g\text{goal8Tile} (\text{BDS} (n:_)) = \text{elems} \ n == \text{elems} \ g8T
\]

Putting things together:
A depth-first search producing the first sequence of moves (in reverse order) that lead to the goal configuration:

\[
\text{dfs8Tile} :: [[\text{Position}]] \\
\text{dfs8Tile} = \text{map} \ \text{elems} \ \text{ls} \\
\text{where} \ ((\text{BDS} \ \text{ls}):_) \\
\text{= searchDfs} \ \text{suc8Tile} \ \text{goal8Tile} (\text{BDS} \ [s8T])
\]
Chapter 3.3
Priority-first Search
Priority-first Search

Given:
Let $P$ be a problem specification.

Solving $P$ – The Idea

- Similar to backtracking search, i.e., search for a particular solution of the problem by a systematic trial-and-error exploration of the solution space but order the candidate nodes according to the most promising node (priority-first search/best-first search).

Note: In contrast to plain backtracking search, which proceeds unguided and can thus be considered blind, priority-first search/best-first search benefits from (hopefully correct) information pointing it towards the “more promising” nodes.
Priority-first Search (Cont’d)

Main Problem Characteristics for Applicability

- 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 “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, the solution.
Illustrating Search Strategies

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

- Depth-first search: [1, 2, 5, 4, 6, 3]
- Breadth-first search: [1, 2, 6, 3, 5, 4]
- Priority-first search: [1, 2, 3, 5, 4, 6]

Implementing Priority-first Search as HOF (1)

The Initial Setting:

- A problem with
  - problem instances of kind $p$

and solutions with
  - solution instances of kind $s$

Objective:

- A higher-order function (HOF) $\text{searchPfs}$
  - solving suitably parameterized problem instances of kind $p$ utilizing the “priority-first/best-first” principle.
Implementing Priority-first Search as HOF (2)

Assumptions:
- an acyclic implicit graph
- all solutions shall be computed (not just the first one)

Note: The HOF can be adjusted to terminate after finding the first solution.

The ingredients of searchPfs:
- node: A type representing node information.
- <=: A comparison criterion for nodes; usually, this is the relator <= of the type class `Ord`. Often, the relator <= can not exactly be defined but only in terms of a plausible heuristic.
- succ :: node -> [nodes]: The function succ yields the list of successors of a node.
- goal :: node -> Bool: The function goal determines if a node is a solution.
Implementing Priority-first Search as HOF (3)

The HOF-Implementation:

```
searchPfs :: (Ord node) => (node -> [node]) -> (node -> Bool) -> node -> [node]

searchPfs succ goal x
  = search' (enPQ x emptyPQ)
  where
    search' q
      | pqEmpty q       = []
      | goal (frontPQ q) = frontPQ q : search' (dePQ q)
      | otherwise
         = let x = frontPQ q
            in search' (foldr enPQ (dePQ q) (succ x))
```
The Abstract Data Type PQueue (1)

The user-visible interface specification of the Abstract Data Type (ADT) priority queue PQueue:

```haskell
module PQueue (PQueue,emptyPQ,pqEmpty,enPQ,dePQ,frontPQ) where

emptyPQ :: PQueue a
pqEmpty :: PQueue a -> Bool
enPQ    :: (Ord a) => a -> PQueue a -> PQueue a
dePQ    :: (Ord a) => PQueue a -> PQueue a
frontPQ :: (Ord a) => PQueue a -> a
```
The Abstract Data Type PQueue (2)

A user-invisible implementation of PQueue as an algebraic data type:

```haskell
newtype PQueue a = PQ [a]

emptyPQ = PQ []

pqEmpty (PQ []) = True
pqEmpty _ = False

enPQ x (PQ q) = PQ (insert x q)
    where insert x [] = [x]
          insert x r@(e:r') | x <= e = x:r
                               | otherwise = e:insert x r'

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

frontPQ (PQ []) = error "frontPQ: empty priority queue"
frontPQ (PQ (x:_)) = x
```
Typical Applications of Priority-first Search

Typical Applications:

- Application areas such as
  - game strategies
  - ...
- The eight-tile problem (8TP)
- ...

- (continues)
A Priority-first Search for 8TP

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

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

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

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

pfs8Tile :: [[Position]]
pfs8Tile = map elems ls
  where ((BDS ls)_ )
    = searchPfs succ8Tile goal8Tile (BDS [s8T])
Chapter 3.4
Greedy Search
Greedy Search

Given:
Let \( P \) be a problem specification.

Solving \( P \) – The 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 (Cont’d)

Main Problem Characteristics for Applicability

- 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.
- 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 reasonable giving a “good,” not necessarily optimal solution. Greedy search then becomes a heuristic algorithm.
Illustrating Greedy Search

Successive stages in a greedy algorithm:

Possible choice

Selected choice

Candidate node

Step 1

Step 2

Step 3

This path constitutes the "optimal" solution

Implementing Greedy Search as HOF (1)

The Initial Setting:

- A problem with
  - problem instances of kind $p$

  and solutions with
  - solution instances of kind $s$

Objective:

- A higher-order function (HOF) $\text{searchGreedy}$
  - solving suitably parameterized problem instances of kind $p$ utilizing the “greedy/hill climbing” principle.
Implementing Greedy Search as HOF (2)

Assumptions:
▶ an acyclic implicit graph
▶ no local minimums, i.e., no locally best solutions

The ingredients of searchGreedy:
▶ node: A type representing node information.
▶ <=: A comparison criterion for nodes; usually, this is the relator <= of the type class Ord.
▶ succ :: node -> [nodes]: The function succ yields the list of successors of a node.
▶ goal :: node -> Bool: The function goal determines if a node is a solution.
Implementing Greedy Search as HOF (3)

The HOF-Implementation:

searchGreedy ::
  (Ord node) => (node -> [node]) -> (node -> Bool)
  -> node -> [node]

searchGreedy succ goal x
  = search’ (enPQ x emptyPQ)
    where
      search’ q
        | pqEmpty q        = []
        | goal (frontPQ q) = [frontPQ q]
        | otherwise
        = let x = frontPQ q
            in search’ (foldr enPQ emptyPQ (succ x))
Implementing Greedy Search as HOF (4)

**Note:**

- The most striking difference to the HOF `searchPfs` is the replacement of `dePQ q` by `emptyPQ` in the recursive call to `search’` to remove old candidate nodes from the priority queue.
Typical Applications of Greedy Search

Typical Applications:

- Graph algorithms, e.g., Prim’s minimum spanning tree algorithm
- Money Change Problem (MCP)
- ...
A Greedy Search for MCP

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

Modeling coins:

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

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

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

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

succCoins :: NodeChange -> [NodeChange]
succCoins (r,p)
  = [ (r-c,c:p) | c <- coins, r-c >= 0 ]
A Greedy Search for MCP (Cont’d)

The goal function:

\[
goalCoins :: \text{NodeChange} \rightarrow \text{Bool} \\
goalCoins (v, _) = v == 0
\]

Putting things together:

\[
\text{change} :: \text{Int} \rightarrow \text{SolChange} \\
\text{change amount} \\
\quad = \text{snd} \ (\text{head} \ (\text{searchGreedy} \ \text{succCoins} \ \text{goalCoins} \\
\quad \quad \quad \quad \quad \quad (\text{amount}, [])))
\]

Example: \( \text{change 199} \rightarrow [2,2,5,20,20,50,100] \)

Note: For \( \text{coins} = [1,3,6,12,24,30] \) the above algorithm can yield suboptimal solutions: E.g., \( \text{change 48} \rightarrow [30,12,6] \) instead of the optimal solution \( [24,24] \).
Chapter 3.1–3.4: Further Reading (1)


Chapter 3.1–3.4: Further Reading (2)

1. Jon Kleinberg, Éva Tardos. Algorithm Design. Addison-Wesley/Pearson, 2006. (Chapter 4, Greedy Algorithms; Chapter 5, Divide and Conquer)


Chapter 3.1–3.4: Further Reading (3)


Chapter 3.5
Dynamic Programming
Dynamic Programming

Given:

Let $P$ be a problem specification.

Solving $P$ – The 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 sections might suffer from generating a large number of identical subproblems. This replication of work can severely impair performance. Dynamic programming aims at overcoming this shortcoming by systematically precomputing and reusing results in a bottom-up fashion, i.e., from smaller to larger problem instances.
Illustrating Dynamic Programming for fib

The dynamic programming computation of the Fibonacci numbers (no recomputation of the solution to subproblems!):

```
```
Illustrating Divide-and-Conquer for fib

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

```
<table>
<thead>
<tr>
<th>fib 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib 3</td>
</tr>
<tr>
<td>fib 2</td>
</tr>
<tr>
<td>fib 2</td>
</tr>
<tr>
<td>fib 1</td>
</tr>
<tr>
<td>fib 1</td>
</tr>
<tr>
<td>fib 0</td>
</tr>
<tr>
<td>fib 1</td>
</tr>
<tr>
<td>fib 0</td>
</tr>
</tbody>
</table>
```

Fethi Rabhi, Guy Lapalme.


Addison-Wesley, 1999, page 179.
Implementing Dynamic Programming as HOF (1)

The Initial Setting:

- A problem with
  - problem instances of kind $p$

and solutions with

- solution instances of kind $s$

Objective:

- A higher-order function (HOF) $\text{dynamic}$
  - solving suitably parameterized problem instances of kind $p$ utilizing the “dynamic programming” principle.
Implementing Dynamic Programming as HOF (2)

The ingredients of the HOF `dynamic`:

- `compute :: (Ix coord) => Table entry coord -> coord -> entry`: Given a table and an index, the function `compute` computes the corresponding entry in the table (possibly using other entries in the table).

- `bnds :: (Ix coord) => (coord,coord)`: The parameter `bnds` represents 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) -> (coord,coord) -> (Table entry coord)

dynamic compute bnds = t
  where
    t = newTable (map (\coord -> (coord,compute t coord)) (range bnds))
```
The Abstract Data Type Table (1)

The user-visible interface specification of the Abstract Data Type (ADT) Table:

module Table (Table,newTable,findTable,updTable)
where

newTable :: (Ix b) => [(b,a)] -> Table a b
findTable :: (Ix b) => Table a b -> b -> a
updTable :: (Ix b) => (b,a) -> Table a b
            -> Table a b

Note:

- The function newTable takes a list of (index, value) pairs and returns the corresponding table.
- The functions findTable and updTable are used to retrieve and update values in the table.
The Abstract Data Type Table (2)

A user-invisible implementation of Table as an Array:

```haskell
newtype Table a b = Tbl (Array a b)

newTable l = Tbl (array (lo,hi) l)
  where indices = map fst l
        lo = minimum indices
        hi = maximum indices

findTable (Tbl a) i = a ! i

updTable p@(i,x) (Tbl a) = Tbl (a // [p])
```

Note:

- The function `newTable` determines the boundaries of the new table by computing the maximum and the minimum key in the association list.
- In the function `findTable`, access to an invalid key returns a system error, not a user error.
Typical Applications of Dynamic Programming

Typical Applications:

- Fibonacci numbers
- Chained matrix multiplication
- Optimal binary search (in trees)
- The travelling salesman problem
- Graph algorithms, e.g., all-pairs shortest path
Computing Fibonacci Numbers using Dynamic Programming

Defining the problem-dependent parameters:

\[
\text{bndsFibs :: Int} \to \text{(Int,Int)}
\]
\[
bndsFibs \ n = (0, n)
\]

\[
\text{compFib :: Table Int Int} \to \text{Int} \to \text{Int}
\]
\[
\text{compFib} \ t \ i
\]
\[
| \ i \leq 1 \quad = \quad i
\]
\[
| \ \text{otherwise} \quad = \quad \text{findTable} \ t \ (i-1) + \text{findTable} \ t \ (i-2)
\]

Putting things together:

\[
\text{fib :: Int} \to \text{Int}
\]
\[
\text{fib} \ n = \text{findTable} \ t \ n
\]
\[
\quad \text{where} \ t = \text{dynamic compFib (bndsFibs n)}
\]
Comparing Dynamic Programming and Memoization

Overall

- **Dynamic programming** and **memoization** enjoy very much the same characteristics and offer the programmer quite similar benefits.
- In practice, differences in behaviour are **minor** and strongly **problem-dependent**.
- In general, both techniques are **equally 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).
Comparing Dynamic Programming and Memoization (Cont’d)

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 computing and filling of the argument/result pairs table allows sometimes slightly more efficient (by a constant factor) implementations.
Comparing Dynamic Programming and Memoization (Cont’d)

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 before they are needed.
Chapter 3.5: Further Reading (1)


Chapter 3.5: Further Reading (2)


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

Chapter 3.5: Further Reading (3)

- 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 (4)


Chapter 4
Equational Reasoning
Chapter 4.1
Motivation
Functional Programming

➢ The usage of $=$ in functional definitions of the type

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

as e.g. used in Haskell in the definition of a function $f$ are genuine mathematical equations.

➢ The equations state that the expressions on the left hand side and the right hand side have the same value.
Functional vs. Imperative Programming (2)

Imperative Programming

- The usage of $=$ in imperative languages like C, Java, etc. in (assignment) statements of the form

$$ x = x+y $$

does not mean that $x$ and $x+y$ have the same value.

- Here, $=$ is used to denote 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 programming languages use thus a different notation, e.g. $:\ =$ such as in Pascal, to denote the assignment operator (instead of the conceptually misleading notation $=$).
Consequence

Reasoning about
  ▶ functional definitions
is because of this difference a lot easier as about
  ▶ programs using destructive assignments

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

For example: The sequence of definitions in Haskell

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

raises an error "x" multiply defined since = in Haskell has the meaning “is by definition equal to”; redefinition is forbidden.
Illustrating Algebraic Reasoning

By algebraic reasoning on equations we obtain:

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

Proof:

\[(a + b) \times (a - b) = a \times a - a \times b + b \times a - b \times b\]

(Distributivity of \(\times\), \(+\))

\[(a \times a - a \times b + b \times a - b \times b) = a \times a - a \times b + a \times b - b \times b\]

(Commutativity of \(\times\))

\[= a \times a - b \times b\]

\[= a^2 - b^2\]
Extending Algebraic Reasoning to Functional Definitions

First Example:

This allows us to conclude: 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.
Reasoning on Functional Definitions – More Examples (1)

Second Example:

Let

\[ a = 3 \]
\[ b = 4 \]

\[ f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f \ x \ y = x^2 + y^2 \]

By equational reasoning on the functional definition of \( f \) and those of \( a \) and \( b \) we can show that the Haskell expression

\[ f \ a \ (f \ a \ b) \]

has value 634.
Reasoning on Functional Definitions – More Examples (2)

Proof:

\[
\begin{align*}
    f\ a\ (f\ a\ b) &= f\ a\ (a^2 + b^2) \\
    &= f\ 3\ (3^2 + 4^2) \\
    &= f\ 3\ (9 + 16) \\
    &= f\ 3\ 25 \\
    &= 3^2 + 25^2 \\
    &= 9 + 625 \\
    &= 634
\end{align*}
\]

Note that the (Haskell) expression \( f\ a\ (f\ a\ b) \) is solely evaluated by equational reasoning applying standard algebraic mathematical laws and the Haskell definitions of \( a, b, \) and \( f. \)
Reasoning on Functional Definitions – More Examples (3)

Third Example:

Let

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

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

By **equational reasoning** on the **functional definitions** of \( g \) and \( h \) we can show the equality of the Haskell expressions

\[ h \ (a+b) \ (a-b) \] and \[ g \ a \ b. \]
Reasoning on Functional Definitions – More Examples (4)

Proof:

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

(Unfolding \( h \)) \[ = (a + b) \ast (a - b) \]

(Distributivity of \( \ast, + \)) \[ = a \ast a - a \ast b + b \ast a - b \ast b \]

(Commutativity of \( \ast \)) \[ = a \ast a - a \ast b + a \ast b - b \ast b \]

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

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

(Folding \( g \)) \[ = g \ a \ b \]
Remark (1)

We have:

In equational reasoning functions can be applied/unapplied

- from left-to-right, called unfolding
- from right-to-left, called folding
Remark (2)

**Note:** Some care needs to be taken though. Let

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

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

- can just be viewed as a logical property that can freely be applied in both directions.

The second equation, however, \( isZero \ n = \text{False} \) cannot, since Haskell implicitly imposes an ordering on the equations:

- Application from left-to-right (i.e., replacing \( isZero \ n \) by \( \text{False} \)), and from right-to-left (i.e., replacing \( \text{False} \) by \( isZero \ n \) for some \( n \)) is legal only, if \( n \) is different from \( 0 \).
Reasoning on Functional Definitions – More Examples (5)

Fourth Example:

The standard implementation of the `reverse` function

\[
\text{reverse} :: \ [a] \rightarrow \ [a] \\
\text{reverse} \ [\] = \ [] \\
\text{reverse} \ (x:xs) = \text{reverse} \ xs \ ++ \ [x]
\]

\[
(\++) :: \ [a] \rightarrow \ [a] \rightarrow \ [a] \\
(\++) \ [\] \ ys = \ ys \\
(\++) \ (x:xs) \ ys = x : (xs ++ ys)
\]

requires \(\frac{n(n+1)}{2}\) calls of the concatenation function `(++)`, where \(n\) denotes the length of the argument list.
A more efficient implementation of the functionality of the `reverse` function is

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

Equational reasoning on functional definitions together with inductive proof principles, here structural induction, allows us to prove:

The Haskell expressions

   reverse xs and fastReverse xs

are equal for all finite lists xs.
Summing up

Functional definitions are
genuine mathematical equations.

This allows us to prove

equality and other relations of functional expressions
by applying standard algebraic mathematical reasoning.

In particular, this can be used to replace

less efficient (called specification) by more efficient (called implementation) implementations of some functionality.

Examples:

Basic: Replace \((x*y)+(x*z)\) by \(x*(y+z)\)

Advanced: Replace reverse by fastReverse
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: From reverse to fastReverse.
It is important to note:

The functional pearl

▶ is not the final (efficient) implementation
▶ but the calculation process leading to it!
Functional Pearls – Origin and Background (1)

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, some

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

and other problems.

Some more appeared in proceedings of conferences including editions of the

- *International Conference of Functional Programming*
- *Mathematics of Program Construction*
Functional Pearls – Origin and Background (3)

Roughly,

- a quarter of these pearls have been written by Richard Bird

In his recent monograph


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

In this chapter, we will consider some of these functional pearls for illustration:

- The Smallest Free Number
- Not the Maximum Segment Sum
- A Simple Sudoku Solver
Last but not least

It is worth noting:

The name of the functional programming language

- GoFER

is an acronym 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**!
Simple Algorithm for solving the SFN-Problem

This suggests the following simple algorithm for solving the SFN-problem:

The simple SFNP-Algorithm:

1. Represent $X$ as a list of integers $x_s$.
2. Sort $x_s$ increasingly, while removing all duplicates.
3. Compute the first gap in the list obtained from the previous step.
Possible Implementation of the Simple Algorithm

...by means of a system of functions

- **ssfn** (reminding to “simple sfn”) and
- **sap** (reminding to “search and pick”)

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

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

The simple SFNP-Algorithm is sound but inefficient:

- Sorting is not of linear time complexity.

The Advanced SFNP-Algorithm Problem:

Develop an algorithm LinSFNP for solving the SFN-problem that is of

- linear time complexity, i.e., that 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 be specified as a function \texttt{minfree}, defined by

\[
\text{minfree} :: [\text{Nat}] \to \text{Nat} \\
\text{minfree} \ \text{xs} = \text{head} \ (\ [0..]) \ \setminus \ \text{xs}
\]

with

\[
(\setminus) :: \text{Eq} \ a \to [a] \to [a] \\
\text{xs} \ \setminus \ \text{ys} = \text{filter} \ (\not\!\text{Elem} \ \text{ys}) \ \text{xs}
\]

denoting difference on sets (i.e., \text{xs} \setminus \text{ys} is the list of those elements of \text{xs} that remain after removing any elements in \text{ys}) and

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

the type of natural numbers starting from 0.
Analysing \texttt{minfree}

The function \texttt{minfree} solves the \texttt{SFN}-problem but its evaluation requires on a list of length $n$

- $\Theta(n^2)$ steps in the worst case.

For illustration consider:

Evaluating

- \texttt{minfree} $[n-1,n-2\ldots 0]$ requires evaluating
  
  $i$ is not an element in $[n-1,n-2\ldots 0]$

  for $0 \leq i \leq n$, and thus $n(n + 1)/2$ equality tests.
Outline

Starting from \texttt{minfree} we will develop an
\begin{itemize}
\item array based and a
\item divide-and-conquer based
\end{itemize}
linear time algorithm for the SFN-problem.

The \textbf{key fact (KF)} both algorithms rely on is:
\begin{itemize}
\item There is a number in $[0..\text{length } \texttt{xs}]$ that is not in \texttt{xs}
\end{itemize}
where \texttt{xs} denotes the initial list of natural numbers.

This implies:
\begin{itemize}
\item The smallest number not in $\text{filter (<=n) xs}$,
$n == \text{length } \texttt{xs}$, is the smallest number not in \texttt{xs}!
Towards the Array-Based Algorithm

The array-based algorithm uses KF to build a

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

The checklist is a

- Boolean array with \( n + 1 \) slots, numbered from 0 to \( n \), whose initial entries are set to False.

Algorithmic idea:

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

The array-based algorithm LinSFNP:

\[
\text{minfree} = \text{search} \cdot \text{checklist}
\]

\[
\begin{aligned}
\text{search} &: \text{Array Int Bool} \to \text{Int} \\
\text{search} &= \text{length} \cdot \text{takeWhile id} \cdot \text{elems}
\end{aligned}
\]

\[
\begin{aligned}
\text{checklist} &: [\text{Int}] \to \text{Array Int Bool} \\
\text{checklist} \, \text{xs} &= \text{accumArray} \, (\|\|) \, \text{False} \, (0, n) \\
&\quad \quad \left( \text{zip} \, (\text{filter} \, (\leq n) \, \text{xs}) \, (\text{repeat} \, \text{True}) \right) \\
&\quad \quad \text{where n} = \text{length} \, \text{xs}
\end{aligned}
\]

Note: This algorithm
- does not require the elements of \text{xs} to be distinct
- but does require them to be natural numbers
Two Variants of the Array-Based Algorithm (1)

1st Variant: 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.

This allows

- replacing of `checklist` by `countlist`.

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

```haskell
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.
Two Variants of the Array-Based Algorithm (2)

2nd Variant: Instead of using a smart library function as in the 1st variant, `checklist` can be implemented

- using a constant-time array update operation.

In Haskell, this can be done using a suitable `monad`, such as 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: This variant is essentially

- a procedural program in functional clothing.
Towards the Divide-and-Conquer Algorithm (1)

Algorithmic idea:

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

First, we collect some properties satisfied by the set difference operation:

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

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 also have:

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

Going on, choose any natural number \( b \), and let

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

then

- \( as \) and \( vs \) are disjoint, and \( bs \) and \( us \) are disjoint.

This implies:

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

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

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

Moreover, because of

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

we obtain (still for any natural number b):

The Basic Divide-and-Conquer Algorithm:

\[
\text{minfree } \text{xs} = \text{if } (\text{null } ([0..b-1]) \setminus \text{us}) \\
\text{then } (\text{head } ([b..]) \setminus \text{vs}) \\
\text{else } (\text{head } ([0..]) \setminus \text{us}) \\
\text{where } (\text{us},\text{vs}) = \text{partition } (\langle b \rangle) \text{xs}
\]
Refining the Divide-and-Conquer Algorithm (1)

Note, the straightforward evaluation of the test

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

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

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

This allows us to replace the test by a test on the length of us:

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

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

divide-and-conquer algorithm.
Refining the Divide-and-Conquer Algorithm (2)

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

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

where every element of \texttt{xs} is assumed to be

\begin{itemize}
  \item greater than or equal to \texttt{a}.
\end{itemize}
Provided \( b \) is chosen so that both

- \( \text{length } \text{us} \) and \( \text{length } \text{vs} \) are less than \( \text{length } \text{x}\)

the below recursive definition of \text{minfree} is well-founded:

\[
\begin{align*}
\text{minfree } \text{x}\ &= \text{minfrom 0 } \text{x}\ \\
\text{minfrom } \text{a } \text{x}\ &| \ \text{null } \text{x}\ = \text{a} \\
\ &| \ \text{length } \text{us} == \text{b-a} = \text{minfrom } \text{b } \text{vs} \\
\ &| \ \text{otherwise} = \text{minfrom } \text{a } \text{us} \\
\text{where (us,vs)} & = \text{partition (</b) x}
\end{align*}
\]
Refining the Divide-and-Conquer Algorithm (4)

It remains to choose \( b \).

This choice shall ensure:
  
  ▶ \( b > a \)
  
  ▶ The maximum of the lengths of \( us \) and \( vs \) is minimum.

This is achieved by choosing \( b \) as

\[
b = a + 1 + n \div 2
\]

where \( n = \text{length } xs \).
Refining the Divide-and-Conquer Algorithm (5)

If \( n \neq 0 \) and \( \text{length } u_\text{s} < b - a \), then
\[
\text{▶ } (\text{length } u_\text{s}) \leq (\text{n div 2}) < \text{n}
\]

And, if \( \text{length } u_\text{s} = b - a \), then
\[
\text{▶ } (\text{length } v_\text{s}) = (\text{n} - (\text{n div 2}) - 1) \leq \text{n div 2}
\]

With this choice, the number of steps for evaluating

\[
\text{minfrom 0 } x_\text{s}
\]

is \text{linear} in the number of elements of \( x_\text{s} \).
The Optimized Divide-and-Conquer Algorithm

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-and-Conquer Algorithm:

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

The optimized divide-and-conquer algorithm is about

- twice as fast as the incremental array-based program, and
- 20% faster than the accumArray-based program.

It is worth noting, the SFN-problem is not artificial:

- It can be considered a simplification of the common programming task to find some object not in use: Numbers then name objects, and \( X \) the set of objects that are currently in use.
**Summing up (Cont’d)**

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 explains

- why there sometimes seems to be a logarithmic gap between the best functional and the best procedural solutions to a problem.

Sometimes, however, this gap

- vanishes as for the SFN-problem.
Chapter 4.4

Not the Maximum Segment Sum
Background and Motivation

A segment of a list

- is a contiguous subsequence.

The Maximum Segment Sum (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

- $3$ (from the segment $[2, 1]$).
The **MSS**-problem

- had been 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 pearl, however,

- we consider the "**Maximum Non-Segment Sum (MNSS) Problem**".
The Maximum 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 Maximum Non-Segment Sum (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 (from the non-segment \([2, 1, -1]\)).
What does MNSS qualify a Pearl Problem?

It is worth noting:

Let \( L \) be a list of length \( n \).

- There are \( \Theta(n^2) \) segments of \( L \).
- There are \( \Theta(2^n) \) subsequences of \( L \).

Hence

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

This raises the problem

- Can the maximum non-segment sum be computed in \textit{linear} time?

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

The Specifying (Initial) Solution of the MNSS-Problem:

\[ mnss :: [\text{Int}] \rightarrow [\text{Int}] \]
\[ mnss = \text{maximum} \ . \ \text{map sum} \ . \ \text{nonsegs} \]

**Intuition:**

- First, `nonsegs` computes a list of all non-segments of the argument list,
- `map sum` then computes the sum of all these non-segments, and
- `maximum`, finally, picks those whose sum is maximum.
The Implementation of `nonsegs`

The implementation of the function `nonsegs`

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

relies on the supporting functions

- `extract`
- `markings`

which itself relies on the supporting function

- `booleans`
The Implementation of nonsegs (Cont’d)

The implementation of the supporting functions:

\[
\text{markings} :: [a] \rightarrow [[[a,\text{Bool}][]{]}]
\]
\[
\text{markings } xs = [\text{zip } xs \text{ bs} | \hspace{1cm} \\
\hspace{1cm} \text{bs } \leftarrow \text{booleans } (\text{length } xs)]
\]

\[
\text{booleans } 0 = [[]]
\]
\[
\text{booleans } (n+1) = [b:bs | b \leftarrow [\text{True},\text{False}], \hspace{1cm} \\
\hspace{1cm} \text{bs } \leftarrow \text{booleans } n]
\]

\[
\text{extract} :: [[[a,\text{Bool}][]{]}] \rightarrow [[[a]]]
\]
\[
\text{extract } = \text{map } (\text{map } \text{fst } . \text{filter } \text{snd})
\]
The Implementation of nonsegs (Cont’d)

Intuition underlying the supporting functions:

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.
The Implementation of nonsegs (Cont’d)

The function

\[
\text{nonseg :: [(a,Bool)] } \rightarrow \text{ Bool}, \text{ finally, returns } \text{True}
\]
on a list \text{xms} iff \text{map snd xsm} describes a non-segment
marking (its implementation is given later).

Last but not least:

The Boolean list \text{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, respec-
tively.

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:

\[
data \text{ State} = \text{E} \mid \text{S} \mid \text{M} \mid \text{N}
\]

Intuition:

The 4 states of the above automaton are used as follows:

- **State E (for Empty)**, starting state: if in \text{E}, markings only in the set \( F^* \) have been recognized.
- **State S (for Suffix)**: if in state \text{S}, one or more \text{T}s have been processed; hence, this indicates markings in the set \( F^* T^+ \), i.e., a non-empty suffix of \text{T}s.
- **State M (for Middle)**: if in state \text{M}, this indicates the processing of markings in the set \( F^* T^+ F^+ \), i.e., a middle segment.
- **State N (for Non-segment)**: if in state \text{N}, this indicates the processing of non-segments markings.
The Finite State Automaton (Cont’d)

This allows us to define:

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

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

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

It is worth noting:

- 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.
Towards Deriving the Linear Time Algorithm

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

\[
\begin{align*}
\text{mnss} &= \text{maximum} \rightarrow \text{map} \rightarrow \text{sum} \\
\text{extract} \rightarrow \text{filter} \rightarrow \text{nonseg} \rightarrow \text{markings} \\
\text{extract} &= \text{map} \rightarrow (\text{map} \rightarrow \text{fst} \rightarrow \text{filter} \rightarrow \text{snd}) \\
\text{nonseg} &= (== \text{N}) \rightarrow \text{foldl} \rightarrow \text{step E} \rightarrow \text{map} \rightarrow \text{snd}
\end{align*}
\]

Work plan:

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

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

\[
\texttt{pick} :: \text{State} \rightarrow [a] \rightarrow [[[a]]]
\]

\[
\texttt{pick} \ q = \text{extract} \ . \\
\hspace{1cm} \text{filter} \ ((== \ q) \ . \ \text{foldl} \ \text{step} \ E \ . \ \text{map} \ \text{snd}) \ . \\
\hspace{1cm} \text{markings}
\]

We have:

\[
\rightarrow \ \text{nonsegs} == \text{pick} \ N
\]
Properties of pick

Moreover, we can prove

- either by calculation from the definition of pick q (which is tedious!)
- or by referring to the definition of step

the equalities:

```haskell
pick E xs = [[]]
pick S [] = []
pick S (xs++[x]) = map (++[x])
   (pick S xs) ++ pick E xs
pick M [] = []
pick M (xs++[x]) = pick M xs ++ pick S xs
pick N [] = []
pick N (xs++ys) = pick N xs ++
   map (++[x])
   (pick N xs) ++ pick M xs
```
Deriving the Linear Time Algorithm (2)

Second, we recast the definition of *pick* as an instance of *foldl*.

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

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

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

\[
pickall = \text{foldl} \; \text{step} \; ([[]],[],[[],[]])
\]
\[
\text{step} \; (\text{ess}, \; \text{nss}, \; \text{mss}, \; \text{sss}) \; x
\]
\[
= (\text{ess}, \; \text{map} \; (\text{++}[x]) \; \text{(sss++ess)}, \; \text{mss} \; ++ \; \text{sss}, \; \text{nss} \; ++ \; \text{map} \; (\text{++}[x]) \; \text{(nss++mss)})
\]
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.

By means of 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 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 the Application of the Fusion Law (1)

...in our scenario to the instantiations:

\[
\begin{align*}
f &= \text{tuple} \left( \text{maximum} \ . \ \text{map} \ \text{sum} \right) \\
g &= \text{step} \\
a &= ([[]], [], [], [])
\end{align*}
\]

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:

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

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

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

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

Next, we derive $h$ by investigating each component in turn. This is demonstrated for the fourth component in detail. The reasoning for the three components is similar.
Towards the Application of the Fusion Law (3)

$max$ is used as an abbreviation for $maximum$:

\[
max (map \ sum (nss \ dpl \ map (\++ [x]) (nss \ ++ \ mss)))
\]

\[
= (definition \ of \ map)
max (map \ sum \ nss \ ++ \ map (sum . (\++[x]))(nss \ ++ \ mss))
\]

\[
= (since \ sum . (\++[x]) = (+x) . sum)
max (map \ sum \ nss \ ++ \ map ((+x) . sum) nss \ ++ \ mss))
\]

\[
= (since \ max (xs\++ys) = (max xs) \ max (max ys))
max (map \ sum \ nss) \ max \ max (map ((+x) . sum) (nss\++mss))
\]

\[
= (since \ max \ . \ map (+x) = (+x) . max)
max (map \ sum \ nss) \ max (max (map \ sum \ (nss\++mss)) + x)
\]

\[
= (introducing \ n = max (map \ sum \ nss) \ and
m = max (map \ sum \ mss)
n \ max ((n \ max \ m) + x)
\]
Towards the Application of the Fusion Law (4)

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

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

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

$$mnss = fourth . foldl h (0, -\infty, -\infty, -\infty)$$
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:

\[
\text{mnss } xs \\
= \text{fourth } (\text{foldl } h \ (\text{start } (\text{take 3 xs})) \ (\text{drop 3 xs}))
\]

\[
\text{start } [x,y,z] \\
= (0, \max [x+y+z, y+z, z], \max [x, x+y, y], x+z)
\]
Concluding Remarks (1)

The **MSS** problem goes back to **Bentley**:


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


Concluding Remarks (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
Towards the Specifying Solution (1)

Preliminary definitions:

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

\[
\text{type Matrix } a = \text{[Row } a]\n\text{type Row } a = \text{[a]}
\]

Grid: A \( 9 \times 9 \)-matrix of digits.

\[
\text{type Grid} = \text{Matrix Digit}
\text{type Digit} = \text{Char}
\]

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

\[
\text{digits } = \text{['1'..'9']}\n\text{blank } = (== '0')
\]
Towards the Specifying Solution (2)

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 (3)

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

1. 1st Approach:
   - Construct a list of all correctly completed grids.
   - Then test the input grid against them to identify those whose non-blank entries match the given ones.

2. 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 follow the 2nd approach to define the specifying initial solution of the Sudoku-problem.
Specifying Solution of the Sudoku-Problem (1)

The Specifying (Initial) Solution of the Sudoku-Problem:

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

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

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

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

Intuition:

- **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.
Specifying Solution of the Sudoku-Problem (2)

To represent the set of choices we introduce the data type:

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

This allows us to define the subsidiary functions of `solve`, i.e.,

- choices
- expand
- valid
Specifying Solution of the Sudoku-Problem (3)

The implementation of choices:

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

Intuition:

- If the cell is blank, then all digits are installed as possible choices.
- Otherwise there is no choice and a singleton is returned.
Specifying Solution of the Sudoku-Problem (4)

The implementation of `expand`:

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

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

Intuition:

- Expansion is a Cartesian product, i.e., a list of lists given by the function `cp`, e.g., `cp[[1,2],[3],[4,5]]` $\rightarrow$ `[[1,3,4],[1,3,5],[2,3,4],[2,3,5]]`
- `map cp` then returns a list of all possible choices for each row.
- `cp . map cp`, finally, installs each choice for the rows in all possible ways.
Specifying Solution of the Sudoku-Problem (5)

The implementation of \textit{valid}:

\begin{verbatim}
valid :: Grid -> Bool
valid g = all nodups (rows g) &&
    all nodups (cols g) &&
    all nodups (boxs g)
\end{verbatim}

\begin{verbatim}
nodups :: Eq a => [a] -> Bool
nodups [] = True
nodups (x:xs) = all (x/=) xs && nodups xs
\end{verbatim}

\textbf{Intuition:}

- A grid is \textit{valid}, if no row, column or box contains duplicates.
Specifying Solution of the Sudoku-Problem (6)

The implementation of rows and columns:

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

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

Intuition:

- rows is the identity function, since the grid is already given as a list of rows.
- columns computes the transpose of a matrix.
Specifying Solution of the Sudoku-Problem (7)

The implementation of `boxes`:

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

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

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

**Intuition:**
- `group` splits a list into groups of three.
- `ungroup` takes a grouped list and ungroups it.
- `group . map group` produces a list of matrices; transposing each matrix and ungrouping them yields the boxes.
Specifying Solution of the Sudoku-Problem (8)

Illustrating the action of boxs for the 4 × 4-case, when group splits a list into groups of two:

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

Note:

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

Instead of

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

the present approach has gone for functions that

- treat the 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

Example:

- The 3 laws (A), (B), and (C) hold on arbitrary $N \times N$-matrices, in particular on $9 \times 9$-grids:

  rows . rows = id \hspace{1cm} (A)
  cols . cols = id \hspace{1cm} (B)
  boxs . boxs = id \hspace{1cm} (C)

  This means, all 3 functions are involutions.

- The 3 laws (D), (E), and (F) hold on $N^2 \times N^2$-matrices:

  map rows . expand = expand . rows \hspace{1cm} (D)
  map cols . expand = expand . cols \hspace{1cm} (E)
  map boxs . expand = expand . boxs \hspace{1cm} (F)
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!
Towards a Better Performing Algorithm

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

$\text{prune} :: \text{Matrix Choices} \rightarrow \text{Matrix Choices}$

that satisfies

$\text{filter valid . expand} = \text{filter valid . expand . prune}$

and realizes the above idea.
Towards defining prune

Pruning a row

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

where

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

Intuition:

- \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

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

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

We can prove:

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

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.

This requires some reasoning which we exemplify for the boxs case.
Reasoning in the boxes Case (1)

\[
\begin{align*}
\text{filter (all nodups . boxes) . expand} \\
= \{(H), \text{ since boxes . boxes} = \text{id}\} \\
\text{map boxes . filter (all nodups) . map boxes . expand} \\
= \{(F)\} \\
\text{map boxes . filter (all nodups) . expand boxes} \\
= \{\text{definition of expand}\} \\
\text{map boxes . filter (all nodups) . cp . map cp . boxes} \\
= \{(I), \text{ and map f . map g} = \text{map (f . g)}\} \\
\text{map boxes . cp . map (filter nodups . cp) . boxes} \\
= \{(G)\} \\
\text{map boxes . cp . map (filter nodups . cp . pruneRow) . boxes}
\end{align*}
\]
Reasoning in the boxes Case (2)

\[ = \{(l)\} \]
\[ \text{map boxes . filter (all nodups) . cp .} \]
\[ \text{map cp . map pruneRow . boxes} \]
\[ = \{\text{definition of expand}\} \]
\[ \text{map boxes . filter (all nodups) . expand .} \]
\[ \text{map pruneRow . boxes} \]
\[ = \{(H) \text{ in the form map f . filter p} = \text{filter (p . f) . map f}\} \]
\[ \text{filter (all nodups . boxes) . map boxes . expand .} \]
\[ \text{map pruneRow . boxes} \]
\[ = \{(F)\} \]
\[ \text{filter (all nodups . boxes) . expand . boxes .} \]
\[ \text{map pruneRow . boxes} \]
Summing up

► We have shown:

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

where

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

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

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

where

\[
\text{prune}
\]
\[=
\text{pruneBy boxs} \ . \ \text{pruneBy cols} \ . \ \text{pruneBy rows}
\]
2nd and Improved Implementation of solve

The Pruning-improved Implementation of solve:

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

Idea

- Combine pruning with expanding the choices for a single cell only at a time:

\[ \leadsto \text{single-cell expansion} \]

To this end we replace the function `expand` by a new version

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

where `expand1` (defined next) expands the choices of a single cell only.
Towards defining expand1

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.)
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 row rows2 below it:

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

break p xs
  = (takeWhile (not . p) xs, dropWhile (not . p) xs)
```
Remarks on expand1

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

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

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

This suggests:

A matrix is

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

It is worth noting:

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

Completeness and safety can be tested as follows.

- **Completeness Test:**
  
  \[
  \text{complete} = \text{all} \ (\text{all single})
  \]
  
  where `single` is the test for a singleton list.

- **Safety Test:**

  \[
  \text{safe m} = \text{all ok (rows m) } \&\& \text{ all ok (cols m) } \&\& \text{ all ok (boxs m)}
  \]

  where

  \[
  \text{ok row} = \text{nodups} \ [d \mid [d] \leftarrow \text{row}]
  \]
We can show

If a matrix is safe but incomplete, we can calculate:

\[
\text{filter valid . expand} = \{ \text{since } \text{expand} = \text{concat . map expand} . \text{expand1} \}
\]

on incomplete matrices

\[
\text{filter valid . concat . map expand} . \text{expand1} = \{ \text{since } \text{filter p . concat} = \text{concat . map (filter p)} \}
\]

\[
\text{concat . map (filter valid . expand)} . \text{expand1} = \{ \text{since } \text{filter valid . expand} = \text{filter valid . expand . prune} \}
\]

\[
\text{concat . map (filter valid . expand . prune)} . \text{expand1}
\]
3rd and Final Implementation of \texttt{solve}

Introducing

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

we have on \texttt{safe} but \texttt{incomplete} matrices that

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

This allows:

The Final Implementation of \texttt{solve}:

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

\[
\begin{align*}
\text{search \ m} & \\
& | \ \text{not (safe \ m)} = [] \\
& | \ \text{complete \ m'} = [\text{map (map head) m'}] \\
& | \ \text{otherwise} = \text{concat (map search (expand1 m'))} \\
\end{align*}
\]

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: 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
Objective

How can we gain (sufficiently much) confidence that

▶ ours and
▶ other people’s programs
are sound?

Essentially, there are two means at our disposal:

▶ Verification
▶ Testing
Verification vs. Testing

▶ Verification
  ▶ Formal soundness proof (soundness of the specification, soundness of the implementation).
  ▶ High confidence but often high effort.

▶ Testing
  ▶ Two Variants
    ▶ Ad hoc: Controllable effort but usually unquantifiable, questionable quality statement.
    ▶ Systematically: Controllable effort with quantifiable quality statement.
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

On the other hand, testing is often

- amazingly successful in revealing errors.
Minimum Requirements of Testing

(Systematic) testing of programs should be

- Specification-based
- Tool-supported
- Automatically
Minimum Requirements of Testing (Cont’d)

There shall be reporting on

► What has been tested?
► How thoroughly, how comprehensively has been tested?
► How was success defined?

Desirable, too

► Reproducibility of tests
► Repeated testing after program modifications
Program Specification

Inevitable

- **Specification** of the meaning of the program
  - **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)
    - advantage: precise and rigorous, unambiguous
In this chapter

Specification-based, tool-supported testing in Haskell with QuickCheck:

- **QuickCheck** (a combinator library)
  - defines a **formal specification language**
    ...that allows property definitions inside of the (Haskell) source code.
  - defines a **test data generator language**
    ...that allows a simple and concise description of a large number of tests.
  - allows **tests** to be **repeated at will**
    ...which ensures reproducibility.
  - allows **automatic testing** of all properties specified in a module, including failure reports
    ...that are automatically generated.
QuickCheck and its specification and test data generator languages are:

- Examples of so-called domain-specific embedded languages
  ↞ special strength of functional programming.

- Implemented as a combinator library in Haskell
  ↞ allows us to make use of the full expressiveness of Haskell when defining properties and test data generators.

- Part of the standard Haskell-distribution (for both GHC and Hugs; see module QuickCheck)
  ↞ ensures easy and direct usability.
Chapter 5.1

Property Definitions
Simple Property Definition w/ QuickCheck (1)

In the simplest cases properties are defined in terms of predicates, i.e., as Boolean valued functions.

Example:

Define inside of the program the property

\[
\text{prop\_PlusAssociative} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}\\
\text{prop\_PlusAssociative} \ x \ y \ z = (x+y)+z == x+(y+z)
\]

Double-checking the property with Hugs yields:

Main>quickCheck prop\_PlusAssociative
OK, passed 100 tests
Simple Property Definition w/ QuickCheck (2)

Note:

- The type specification for `prop_PlusAssociative` is required because of the overloading of `+` (otherwise there will be an error message on ambiguous overloading: `QuickCheck` needs to know which test data to generate).

- The type specification allows a type-specific generation of test data.
Simple Property Definition w/ QuickCheck (3)

The same example slightly varied:

Define inside of the program the property

\[
\text{prop\_PlusAssociative :: Float -> Float -> Float -> Bool}
\]

\[
\text{prop\_PlusAssociative x y z = (x+y)+z == x+(y+z)}
\]

Double-checking the property with Hugs yields:

```
Main>quickCheck prop_PlusAssociative
Falsifiable, after 13 tests:
1.0
-5.16667
-3.71429
```
Simple Property Definition w/ QuickCheck (4)

Note:

- The property is **falsifiable** for type `Float`: think e.g. of rounding errors.

The error report contains:

- The number of tests successfully passed
- A counter example
Advanced Property Definition (1)

Given:
- A function `insert`
- A predicate `ordered`

Property under test:
- Insertion into a sorted list

A straightforward property definition to double-check the correctness of the insertion function were:

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

However, this property is falsifiable.
- The definition is naive and too strong
  (note that `xs` is not supposed to be sorted).
Advanced Property Definition (2)

First fix (trial-and-error):

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

Note:

- **ordered xs ==>**: This adds a **precondition** to the property definition.
  - ~⇒~ Generated test data that do not match the precondition, are dropped.

- **==>**: is **not** a simple Boolean operator but affects the selection of test data.
  - ~⇒~ Property definitions that rely on such operators always have the result type **Property** in **QuickCheck**.

- **Overall**: A trial-and-error approach to generating test data: Generate, then check if usable; if not, drop.
Advanced Property Definition (3)

Second fix (systematic):

```haskell
prop_InsertOrdered :: Int -> Property
prop_InsertOrdered x =
    forAll orderedLists $ \xs -> ordered (insert x xs)
```

Note:

- This fix works by **direct quantifying** (in the running example: direct quantifying over sorted lists)
- **Overall**: A **systematic** approach to generating test data: Only useful test data are generated.
The Operator ($$\text{}$$) — A Quick Reminder

Standard Prelude:

\[
(\text{	extdollar}) : : (a \rightarrow b) \rightarrow a \rightarrow b
\]
\[
f \text{	extdollar} \ x = f \ x
\]

Remark:

- The operator ($$\text{}$$) is Haskell’s *infix function application*.
- It is useful to avoid the usage of parentheses:
  
  Example: \( f \ (g \ x) \) can be written as \( f \ \text{	extdollar} \ g \ x \).
Note

Expressiveness:

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

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

Testing multiple properties:

A (small) program (also called quickCheck) can be run from the command line

- >quickCheck Module.hs

in order to test all properties defined in Module.hs at once.
Chapter 5.2
Testing against Abstract Models
Idea

Testing the correctness of an implementation against a reference implementation, a so-called

- abstract model (reference model)

In the following:

- Demonstrating this by an extended example: Developing an abstract data type for queues.
Abstract Model of Queues

An abstract data type for first-in-first-out (FIFO) queues.

Specification:

```hs
type Queue a = [a]
empty = []
add x q = q ++ [x] -- inefficient due to ++!
isEmpty q = null q
front (x:q) = x
remove (x:q) = q
```

This is a simple (but inefficient) implementation that we consider the abstract model of a FIFO queue; it is our reference model of a FIFO queue.
The Concrete Model of Queues (1)

...the implementation of interest, a more efficient implementation than the one of the abstract model.

Basic Idea:

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

Together this ensures:

- Efficient access to list front and list back
  \[ \Rightarrow ++ \text{ for addition boils down to: (strength reduction)} \]

Example:

- Abstract queue: \([7, 2, 9, 4, 1, 6, 8, 3]++[5]\)
- Possible concrete queues:
  - \(([7, 2, 9, 4], 5: [3, 8, 6, 1])\)
  - \(([7, 2], 5: [3, 8, 6, 1, 4, 9])\)
  - ...

\[368/1368\]
The Concrete Model of Queues (2)

Implementation:

```
type QueueI a = ([a],[a])
emptyI = ([],[])
addI x (f,b) = (f,x:b)
isEmptyI (f,b) = null f
frontI (x:f,b) = x
removeI (x:f,b) = flipQ (f,b)

where
  flipQ ([],b) = (reverse b, [])
  flipQ q = q
```
In the following

We think of
  ▶ Queue and
  ▶ QueueI
in terms of
  ▶ specification and
  ▶ implementation
of FIFO queues, respectively.

Next we want to double-check/test if operations defined on QueueI (implementation / queues) behave in the same way as the operations defined on Queue (specification / abstract queues).
Relating Queues and Abstract Queues

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

```haskell
retrieve :: QueueI Integer -> [Integer]
retrieve (f,b) = f ++ reverse b
```

The function `retrieve`

- transforms each of the (usually many) concrete representations, i.e., values of `QueueI`, of an abstract queue, i.e., a value of `Queue`, into their unique canonical representation of an abstract queue.
Soundness Properties for Operations on QueueI

The understanding of QueueI and Queue as lists on integers

- allows us to omit type specifications in the definitions of properties defined next.

By means of retrieve we can double-check, if

- the results of applying the efficient operations on QueueI coincide with those of the abstract operations on Queue.
Soundness Properties: Initial Definitions (1)

The below properties can reasonably be expected to hold:

\[
\begin{align*}
\text{prop_empty} & = \text{retrieve emptyI} == \text{empty} \\
\text{prop_add} x q & = \text{retrieve (addI} x q) \\
& \quad == \text{add} x \ (\text{retrieve} q) \\
\text{prop_isEmpty} q & = \text{isEmptyI} q == \text{isEmpty (retrieve} q) \\
\text{prop_front} q & = \text{frontI} q == \text{front (retrieve} q) \\
\text{prop_remove} q & = \text{retrieve (removeI} q) \\
& \quad == \text{remove (retrieve} q)
\end{align*}
\]

However, this is not true!
Soundness Properties: Initial Definitions (2)

Testing e.g. `prop_isEmpty` using `QuickCheck` yields:

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

Problem:

- The specification of `isEmpty` assumes implicitly that the following invariant holds:
  - The front of the list is only empty, if the back of the list is empty, too:
    
    ```
    isEmptyI (f,b) = null f
    ```
Soundness Properties: Initial Definitions (3)

In fact:

- `prop_isEmpty`, `prop_front`, and `prop_remove` are all falsifiable because of this!
- The implementations of `isEmptyI`, `frontI`, and `removeI` assume implicitly that the front of a queue will only be empty if the back also is.

This silent assumption has to be made explicit in terms of an invariant.
Soundness Properties: Refined Definitions (1)

We define the invariant as follows:

\[
\text{invariant} :: \text{QueueI Integer} \rightarrow \text{Bool} \\
\text{invariant} (f,b) = \text{not (null f)} || \text{null b}
\]

...and add them to the relevant property definitions:

\[
\begin{align*}
\text{prop\_empty} &= \text{retrieve emptyI} == \text{empty} \\
\text{prop\_add} x q &= \text{invariant q} \Rightarrow \\
&\quad \text{retrieve (addI x q)} == \text{add x (retrieve q)} \\
\text{prop\_isEmpty} q &= \text{invariant q} \Rightarrow \\
&\quad \text{isEmptyI q} == \text{isEmpty (retrieve q)} \\
\text{prop\_front} q &= \text{invariant q} \Rightarrow \\
&\quad \text{frontI q} == \text{front (retrieve q)} \\
\text{prop\_remove} q &= \text{invariant q} \Rightarrow \\
&\quad \text{retrieve (removeI q)} == \text{remove (retrieve q)}
\end{align*}
\]
Soundness Properties: Refined Definitions (2)

Now, testing `prop_isEmpty` using QuickCheck yields:

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

However, testing `prop_front` still fails:

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

Problem:

- `frontI` (as well as `removeI`) may only be applied to non-empty lists. So far, we did not take care of this.
Soundness Properties: Final Definitions

Fix:

- Add \( \text{not } (\text{isEmptyI } q) \) to the preconditions of the relevant properties.

This leads to:

\[
\begin{align*}
\text{prop_empty} &= \text{retrieve } \text{emptyI } = \text{empty} \\
\text{prop_add } x \ q &= \text{invariant } q \implies \\
&\quad \text{retrieve } (\text{addI } x \ q) = \text{add } x \ (\text{retrieve } q) \\
\text{prop_isEmpty } q &= \text{invariant } q \implies \\
&\quad \text{isEmptyI } q = \text{isEmpty } (\text{retrieve } q) \\
\text{prop_front } q &= \text{invariant } q \&\& \text{not } (\text{isEmptyI } q) \implies \\
&\quad \text{frontI } q = \text{front } (\text{retrieve } q) \\
\text{prop_remove } q &= \text{invariant } q \&\& \text{not } (\text{isEmptyI } q) \implies \\
&\quad \text{retrieve } (\text{removeI } q) = \text{remove } (\text{retrieve } q)
\end{align*}
\]

Now:

- All properties pass the test successfully!
Soundness Considerations Continued

We are not yet done – we still need to check:

▶ Operations producing queues do only produce queues that satisfy this invariant.

Note:

So far we only tested that

▶ operations on queues behave correctly on representations of queues that satisfy the invariant

\[ \text{invariant } (f,b) = \text{not (null } f) \text{ || null } b \]
Adding Missing Soundness Properties (1)

Defining properties for operations producing queues:

\[
\begin{align*}
\text{prop\_inv\_empty} & \quad = \quad \text{invariant empty}I \\
\text{prop\_inv\_add} \ x \ q & \quad = \quad \text{invariant} \ q \implies \\
 & \text{invariant} \ (\text{add}I \ x \ q) \\
\text{prop\_inv\_remove} \ q & \quad = \quad \text{invariant} \ q \land \\
 & \text{not} \ (\text{isEmpty}I \ q) \implies \\
 & \text{invariant} \ (\text{remove}I \ q)
\end{align*}
\]
Adding Missing Soundness Properties (2)

Testing by means of \texttt{QuickCheck} yields:

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

\textbf{Problem:}

- The invariant must hold
  - not only after applying \texttt{removeI},
  - but also after applying \texttt{addI} to the empty list; adding to the back of a queue breaks the invariant in this case.
Soundness Properties: Completed Now!

To overcome the last and final problem:

- Adjust the function `addI` as follows:
  \[
  \text{addI } x \ (f,b) = \text{flipQ } (f,x:b)
  \]
  -- instead of: \[
  \text{addI } x \ (f,b) = (f,x:b)
  \]
  with \text{flipQ} as defined previously.

Now:

- All properties pass the test successfully!
Summing up

In the course of developing this example it turned out:

- **Testing** revealed (only) one bug in the implementation (this was in function `addI`).
- **But**: Several missing preconditions and a missing invariant in the original definitions of properties were found and added.

Both is typical and valuable:

- The additional conditions and invariants are now explicitly given in the program text.
- They add to understanding the program and are valuable as documentation, both for the program developer and for future users (think e.g. of program maintainance!).
Chapter 5.3

Testing against Algebraic Specifications
Testing against **algebraic specifications** is (often) a useful alternative to testing against an **abstract model**.

An algebraic specification

- provides **equational constraints** the operations ought to satisfy.
Algebraic Specifications

For FIFO queues, e.g., we might start with the following algebraic specifications:

\[
\begin{align*}
\text{prop\_isEmpty} \ q & = \text{invariant } q \implies \ \text{isEmptyI} \ q \equiv (q \equiv \text{emptyI}) \\
\text{prop\_front\_empty} \ x & = \text{frontI} (\text{addI} \ x \ \text{emptyI}) \equiv x \\
\text{prop\_front\_add} \ x \ q & = \text{invariant } q \land \not (\text{isEmptyI} \ q) \implies \text{frontI} (\text{addI} \ x \ q) \equiv \text{frontI} \ q \\
\text{prop\_remove\_empty} \ x & = \text{removeI} (\text{addI} \ x \ \text{emptyI}) \equiv \text{emptyI} \\
\text{prop\_remove\_add} \ x \ q & = \text{invariant } q \land \not (\text{isEmptyI} \ q) \implies \text{removeI} (\text{addI} \ x \ q) \equiv \text{addI} \ x \ (\text{removeI} \ q)
\end{align*}
\]
Testing Algebraic Specifications (1)

Testing prop_remove_add using QuickCheck yields:

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

Problem:

- The left hand side, i.e., `removeI (addI x q)`, yields: `([0,0],[[]])`
- The right hand side, i.e., `addI x (removeI q)`, yields: `([0],[0])`
- The queue representations `([0,0],[[]])` and `([0],[0])` are equivalent (representing both the abstract queue `[0,0]`) but are not equal!
Testing Algebraic Specifications (2)

Fix:

- Consider “equivalent” instead of “equal”:
  \[
  q \; \text{'equiv'} \; q' = \text{invariant} \; q \; \&\& \; \text{invariant} \; q' \; \&\& \\
  \text{retrieve} \; q \; == \; \text{retrieve} \; q'
  \]

In fact: Replacing

\[
\text{prop} \_\text{remove} \_\text{add} \; x \; q = \text{invariant} \; q \; \&\& \\
\text{not} \; (\text{isEmptyI} \; q) \; ===> \\
\text{removeI} \; (\text{addI} \; x \; q) \; == \; \text{addI} \; x \; (\text{removeI} \; q)
\]

by

\[
\text{prop} \_\text{remove} \_\text{add} \; x \; q = \text{invariant} \; q \; \&\& \\
\text{not} \; (\text{isEmptyI} \; q) \; ===> \\
\text{removeI} \; (\text{addI} \; x \; q) \; \text{'equiv'} \; \text{addI} \; x \; (\text{removeI} \; q)
\]

yields as desired:

- The test of \text{prop} \_\text{remove} \_\text{add} passes successfully!
Testing Algebraic Specifications (3)

Similar to the setup in Chapter 5.1, we have to check:

- All operations producing queues yield results that are equivalent, if the arguments are.

Example:

For the operation `addI` this can be expressed by:

```
prop_add_equiv q q' x = q 'equiv' q' ==> 
addI x q 'equiv' addI x q'
```
Summing up

Though mathematically sound, the definition of \texttt{prop\_add\_equiv} is inappropriate for fully automatic testing.

We might observe:

Main>quickCheck prop_add_equiv Arguments exhausted after 58 tests.

Problem and background:

- \texttt{QuickCheck} generates the lists \( q \) und \( q' \) randomly.
- Most of the generated pairs of lists will not be equivalent, and hence be discarded for the actual test.
- \texttt{QuickCheck} generates a maximum number of candidate arguments only (default: 1.000), and then stops, possibly before the number of 100 test cases is met.
Outlook

Enhancing usability of QuickCheck by adding support for

- Quantifying over subsets
  - by means of filters
  - by means of generators (type-based, weighted, size controlled,...)
- ...
- Test case monitoring

In the following:
- Illustrating this support by means of examples!
Chapter 5.4
Quantifying over Subsets
Background and Motivation

For QuickCheck holds:

- By default, parameters are quantified over the values of the underlying type
  (e.g., all integer lists)

Often, however, it is required:

- A quantification over subsets of these values
  (e.g., all sorted integer lists)
Quantifying over Subsets

QuickCheck offers several means for achieving this:

Representation of subsets in terms of

- **Boolean functions** that act as a filter for test cases
  - **Adequate**, if many elements of the underlying set are members of the relevant subset, too.
  - **Inadequate**, if only a few elements of the underlying set are members of the relevant subset.

- **generators**
  - A generator of type `Gen a` yields a random sequence of values of type `a`.
  - The property `forall set p` successively checks `p` on randomly generated elements of `set`. 
Support by QuickCheck

For the effective usage of generators QuickCheck supports:

- different variants for the specification of relations such as `equiv`
  - As a Boolean function
    - easy to check equivalence of two values (but difficult to generate values that are equivalent).
  - As a function from a value to a set of related (e.g., equivalent) values (generator!)
    - easy to generate equivalent values (but difficult to check if two values are equivalent).

The latter option will be considered in more detail in the following chapter.
Chapter 5.5
Generating Test Data
Generators

The fundamental function to make a choice:

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

Note:

- The function `choose` generates “randomly” an element of the specified domain.
- `choose (1,n)` represents the set `{1,...,n}`.
- The type `Gen` is a monad (cp. Chapter 11).
Using choose

...we can define `equivQ`:

```
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
```

- Generates a random queue that contains the same elements as `q`.
- The number `k` of elements in the back of the queue will be chosen such that it is properly smaller than the total number of elements of the queue (under the assumption that the total number is different from 0).
Application (1)

This allows us to check that

- generated elements are related, i.e., equivalent.

To this end check:

\[
\text{prop\_EquivQ}\ q = \text{invariant}\ q \implies \\
\forall (\text{equivQ}\ q) \ $ \ q' \rightarrow q \ '\text{equiv}' \ q'
\]

Note:

- Recall that $ means function application. Using $ allows the omission of parentheses (see the $ expression in the example).
- The property which is dual to prop\_EquivQ, i.e., that all related elements can be generated, cannot be checked by testing.
Application (2)

This allows:

- Reformulating the property that \texttt{addI} maps equivalent queues to equivalent queues

\[
\text{prop\_add\_equiv } q \ x = \text{invariant } q \implies \\
\forall (\text{equivQ } q) \ (\forall q' \rightarrow \\
\text{addI } x \ q \ \text{'equiv'} \ \text{addI } x \ q')
\]

Remark:

- Other properties analogously

Next we consider: How to define generators.
Defining Generators

...is eased because of the monadic type of Gen.

It holds:

- `return a` always yields (generates) a and represents the singleton set `{a}`
- `do {x <- s; e}` can be considered the (generated) set `{e | x ∈ s}`
Type-based Generators (1)

...by means of the overloaded generator *arbitrary*, e.g. for the generation of arguments of properties:

**Example 1:**

\[
\text{prop\_max\_le } x \ y = x \leq x 'max' y
\]

is equivalent to

\[
\text{prop\_max\_le } = \text{forAll arbitrary } \( x \rightarrow \\
\text{forAll arbitrary } \( y \rightarrow x \leq x 'max' y
\]

Type-based Generators (2)

Example 2:

The set \{y \mid y \geq x\} can be generated by

\[
\text{atLeast } x = \text{do } \text{diff} \leftarrow \text{arbitrary} \\
\text{return } (x + \text{abs } \text{diff})
\]

because of the equality

\[
\{y \mid y \geq x\} = \{x + \text{abs } d \mid d \in \mathbb{Z}\}
\]

that holds for numerical types.

Note: Similar definitions are possible for other types, too.
...between several generators can be achieved by means of a generator `oneof` that can be thought of as `set union`.

**Example:** Constructing a sorted list

```haskell
orderedLists = do x <- arbitrary
                 listsFrom x

where
  listsFrom x = oneof [return [], do y <- atLeast x
                         liftM (x:) (listsFrom y)]
```

**Underlying intuition:**
- A sorted list is either empty or the addition of a new head element to a sorted list of larger elements.
Weighted Selection (1)

- The **oneof** combinator picks with equal probability one of the alternatives.
- This often has an unduly impact on the test case generation (in the previous example the empty set will be selected too often).
- **Remedy:** A weight function `frequency` that assigns different weights to the alternatives.

```
frequency :: [(Int,Gen a)] -> Gen a
```
Weighted Selection (2)

Application:

\[
\text{listsFrom } x \\
\quad = \text{frequency } [(1, \text{return } []), \\\n\quad \quad (4, \text{do } y \leftarrow \text{atLeast } x \\\n\quad \quad \quad \quad \text{liftM (x:) (listsFrom } y)) ]
\]

- A \texttt{QuickCheck} generator corresponds to a probability distribution over a set, not the set itself.
- The impact of the above assignment of weights is that on average the length of generated lists is 4.
The Type Class Arbitrary

If non-standard generators such as `orderedLists` are used frequently, it is advisable to make this type an instance of type class `Arbitrary`:

```
newtype OrderedList a = OL [a]

instance (Num a, Arbitrary a) => Arbitrary (OrderedList a) where
  arbitrary = liftM OL orderedLists
```

Together with re-defining `insert` with the type

```
insert :: Ord a => a -> OrderedList a -> OrderedList a
```

arguments generated for it will automatically be ordered.
Controlling the Size of Generated Test Data

- This is usually wise for type-based test data generation
- It is explicitly supported by QuickCheck
Controlling the Size of Generated Test Data

Generators that depend on the size can be defined by:

```haskell
sized :: (Int -> Gen a) -> Gen a
    -- For defining size-aware generators

sized $ \n -> do len <- choose (0,n)
    vector len -- Application of sized
                -- in the Def. of the
                -- default list generator

vector n = sequence [arbitrary | i <- [1..n]]
    -- generates random list
    -- of length n

resize :: Int -> Gen a -> Gen a
    -- for controlling the size
    -- of generated values

sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
    -- Application of resize
```
Generators for User-defined Types

Test data generators for

- predefined ("built-in") types of Haskell
  - are provided by QuickCheck
  - for user-defined types, this is not possible
- user-defined types
  - have to be provided by the user in terms of defining a suitable instance of the type class Arbitrary
  - require usually, especially in case of recursive types, to control the size of generated test data
Example: Binary Trees (1)

Consider type \((\text{Tree } a)\):

```haskell
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

The following definition of the test-data generator is obvious:

```haskell
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency [(1,return Leaf),
              (3,liftM3 Branch
               arbitrary arbitrary arbitrary)]
```
Example: Binary Trees (2)

Note:

- The assignment of weights (1 vs. 3) has been done in order to avoid the generation of all too many trivial trees of size 1.

- **Problem:** The likelihood that a generator comes up with a finite tree, is only one third.

  this is because termination is possible only, if all subtrees generated are finite. With increasing breadth of the trees, the requirement of always selecting the “terminating” branch has to be satisfied at ever more places simultaneously.
Example: Binary Trees (3)

Remedy:

- Usage of the parameter \texttt{size} in order to ensure
  - termination and
  - “reasonable” size
  of the generated trees.
Example: Binary Trees (4)

Implementation:

```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.
Example: Binary Trees (5)

Remark:

- shrub is a generator for “small(er)” trees.
- shrub is not bounded 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.
- Defining generators for recursive types must usually be handled specifically as in this example.
Chapter 5.6
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

- an array of monitoring and reporting possibilities.
Usefulness of Test-Data Monitoring

Why is test-data monitoring meaningful?

Reconsider the example of inserting into a sorted list:

\[
\text{prop}\_\text{InsertOrdered} :: \text{Integer} \to [\text{Integer}] \\
\to \text{Property} \\
\text{prop}\_\text{InsertOrdered} \ x \ \text{xs} = \text{ordered} \ \text{xs} \Rightarrow \\
\text{ordered} \ (\text{insert} \ x \ \text{xs})
\]
QuickCheck performs the check of prop_InsertOrdered such that:

- lists are generated randomly
- each generated list will be checked, if it is sorted (used test case) or not (discarded test case)

Obviously, it holds:

- the likelihood that a randomly generated list is sorted is the higher the shorter the list is

This introduces the danger that

- the property prop_InsertOrdered is mostly tested with lists of length one or two
- even a successful test is not meaningful
Test-Data Monitoring using trivial (1)

For monitoring purposes QuickCheck provides a

▶ combinator trivial, where the meaning of “trivial” is user-definable.

Example:

prop_InsertOrdered :: Integer -> [Integer] -> Property
    prop_InsertOrdered x xs = ordered xs ==> trivial (length xs <= 2) $ ordered (insert x xs)

with

Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial)
Test-Data Monitoring using trivial (2)

Observation:

- 91% are too many trivial test cases in order to ensure that the total test is meaningful
- The operator $\Rightarrow$ should be used with care in test-case generators

Remedy:

- User-defined generators
  $\Rightarrow$ as in the example of `prop_InsertOrdered` in Chapter 5.1 ("Second Fix (systematic)").
Test-Data Monitoring using classify (1)

The combinator \texttt{trivial} is

- instance of a more general combinator \texttt{classify}
  \begin{verbatim}
  trivial p = classify p "trivial"
  \end{verbatim}
Test-Data Monitoring using classify (2)

Multiple applications of classify allow an even more refined test-case monitoring:

```haskell
test-Data Monitoring using classify (2)

Multiple applications of classify allow an even more refined test-case monitoring:

```haskell
prop_InsertOrdered x xs = ordered xs =>
    classify (null xs) "empty lists" $
    classify (length xs == 1) "unit lists" $
    ordered (insert x xs)

This yields:

Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
Test-Data Monitoring using \texttt{collect}

Going beyond, the combinator \texttt{collect} allows us to keep track on all test cases:

\begin{verbatim}
prop_InsertOrdered x xs = ordered xs =>
    collect (length xs) $ ordered (insert x xs)
\end{verbatim}

This yields a histogram of values:

\begin{verbatim}
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
5% 3.
\end{verbatim}
Chapter 5.7

Implementation of QuickCheck
On the Implementation of QuickCheck (1)

A glimpse into the implementation:

class Testable a where
    property :: a -> Property

newtype Property = Prop (Gen Result)

instance Testable Bool where
    property b = Prop (return (resultBool b))

instance (Arbitrary a, Show a, Testable b) => Testable (a->b) where
    property f = forAll arbitrary f

instance Testable Property where
    property p = p

quickCheck :: Testable a => a -> IO ()
QuickCheck

- consists in total of about 300 lines of code.
- has initially been presented by Koen Claessen and John Hughes:

Summing up (1)

In general, it holds:

- Formalizing specifications is meaningful (even without a subsequent formal proof of soundness).

Experience shows:

- Specifications provided are often (initially) faulty themselves.
QuickCheck is an effective tool

- to disclose bugs in
  - programs and
  - specifications
with little effort.

- to reduce
  - test costs
while simultaneously
  - testing more thoroughly.
Summing up (3)

Investigations of Richard Hamlet


indicate that

▶ a high number of test cases yields meaningful results even in the case of random testing.

Moreover

▶ The generation of random test cases is often “cheap.”

Hence, there are many reasons advising

▶ the routine usage of a tool like QuickCheck!
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)
Summing up (5)

The presentation of this chapter is closely based on:


For implementation details and applications refer to:

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 6

Verification
Motivation

Though often amazingly effective, testing is limited to

- showing the presence of errors.
  It can not show their absence!

By contrast, verification is able to

- proving the absence of errors!
In this chapter

...we will consider **important proof techniques** for verifying properties of **functional (and other) programs** that may operate on

- **elementary data** such as
  - integers
  - strings
  - ...

- **composed data** (in Haskell: **algebraic data types**) such as
  - trees
  - **lists** (which are **finite** by definition)
  - **streams** (which are **infinite** by definition)
  - ...


Outline of the Proof Techniques

We already considered (cf. Chapter 4):

▶ Equational reasoning

We will consider in this chapter:

▶ Basic inductive proof principles
  ▶ Natural (or mathematical) induction
  ▶ Strong induction
  ▶ Structural induction

▶ Specialized inductive proof principles
  ▶ Induction on lists
  ▶ Induction on streams

▶ Coinduction

▶ Fixed point induction
Before going into details (1)

...it is worth noting:

Though of different rigor, testing and verification are both instances of approaches that aim at

- ensuring the correctness of a program or system.
Before going into details (2)

Conceptually, we can distinguish between approaches that strive for ensuring correctness by

- **Construction**
  \[\mapsto \text{applied a priori/on-the-fly of the program development}\]

- **Checking**
  \[\mapsto \text{applied a posteriori of the program development}\]
  - **Verification**
  - **Testing** (only to a limited extent if not exhaustive)
Before going into details (3)

With this in mind, we may loosely conclude:

- Correctness by Construction
  - Equational Reasoning
- Correctness by Checking
  - Verification
  - Testing
Chapter 6.1
Equational Reasoning – Correctness by Construction
Equational Reasoning

...is sometimes also called

- proof by program calculation.

It has been considered and demonstrated previously. Consider Chapter 4 for details.
Chapter 6.1: Further Reading (1)


Chapter 6.1: Further Reading (2)


Chapter 6.2

Basic Inductive Proof Principles
Basic inductive proof principles are:

- **Natural or mathematical** induction (dtsch. *vollständige Induktion*)
- **Strong** induction (dtsch. *verallgemeinerte Induktion*)
- **Structural** induction (dtsch. *strukturelle Induktion*)
Basic Inductive Proof Principles

Let $P$ be a property; let $S$ be a set of values $s$ that are (inductively) constructed from a set of (structurally simpler) values $\text{subs}(s)$; let $\text{IN}$ denote the set of natural numbers.

The principles of

- **Natural (mathematical) induction**

\[
(P(1) \land (\forall n \in \text{IN}. P(n) \Rightarrow P(n+1))) \Rightarrow \forall n \in \text{IN}. P(n)
\]

- **Strong induction**

\[
(\forall n \in \text{IN}. (\forall m < n. P(m)) \Rightarrow P(n)) \Rightarrow \forall n \in \text{IN}. P(n)
\]

- **Structural induction**

\[
(\forall s \in S. \forall s' \in \text{subs}(s). P(s')) \Rightarrow P(s) \Rightarrow \forall s \in S. P(s)
\]
Note

The **proof principles** of

- natural (mathematical)
- strong
- structural

induction are equally expressive and powerful.
Illustrating Examples

Next we provide some typical examples illustrating the usage of these three basic inductive principles of

- natural (mathematical)
- strong
- structural

induction.
Chapter 6.2.1
Natural Induction
Example A

Theorem (6.2.1.1)

\[
\forall n \in \mathbb{N}. \quad \sum_{i=1}^{n} i = \frac{n \times (n + 1)}{2}
\]

Proof: By means of natural (mathematical) induction.
Proof of Theorem 6.2.1.1 (1)

Base case: $n = 1$. In this case we obtain the desired equality by a straightforward calculation:

\[
\sum_{i=1}^{n} i = \sum_{i=1}^{1} i = 1 = 1 = \frac{1 \times 2}{2} = \frac{1 \times (1 + 1)}{2} = \frac{n \times (n + 1)}{2}
\]
Proof of Theorem 6.2.1.1 (2)

**Inductive case:** Applying the induction hypothesis (IH) once, we obtain as desired:

\[
\begin{align*}
\sum_{i=1}^{n+1} i &= (n + 1) + \sum_{i=1}^{n} i \\
\text{(IH)} &= (n + 1) + \frac{n \times (n + 1)}{2} \\
&= (n + 1) \times \left(\frac{n}{2} + 1\right) \\
&= \frac{(n + 1) \times (n + 2)}{2} = \frac{(n + 1) \times ((n + 1) + 1)}{2} \\
\end{align*}
\]

\[\square\]
Example B

Theorem (6.2.1.2)

\[ \forall n \in \mathbb{N}. \sum_{i=1}^{n} (2 \times i - 1) = n^2 \]

Proof: By means of natural (mathematical) induction.
Proof of Theorem 6.2.1.2 (1)

Base case: \( n = 1 \). In this case we obtain the desired equality by a straightforward calculation:

\[
\sum_{i=1}^{n} (2 \cdot i - 1) = \sum_{i=1}^{1} (2 \cdot i - 1) \\
= 2 \cdot 1 - 1 \\
= 2 - 1 \\
= 1 \\
= 1^2 = n^2
\]
Proof of Theorem 6.2.1.2 (2)

**Inductive case:** Applying the *induction hypothesis* (IH) once, we obtain as desired:

\[
\sum_{i=1}^{n+1} (2 \times i - 1) = 2 \times (n + 1) - 1 + \sum_{i=1}^{n} (2 \times i - 1)
\]

(IH) \quad = \quad (2 \times (n + 1) - 1) + n^2
\]

\[
= \quad 2n + 2 - 1 + n^2
\]

\[
= \quad 2n + 1 + n^2
\]

\[
= \quad n^2 + 2n + 1
\]

\[
= \quad n^2 + n + n + 1
\]

\[
= \quad (n + 1) \times (n + 1) = (n + 1)^2
\]

□
Chapter 6.2.2
Strong Induction
Fibonacci Function

The Fibonacci function is defined by:

\[ fib : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \]

\[ fib(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
fib(n - 1) + fib(n - 2) & \text{otherwise}
\end{cases} \]
Example

Theorem (6.2.2.1)

∀ n ∈ IN₀. \( f(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \)

Proof: By means of strong induction.
Key Idea for proving Theorem 6.2.2.1

Using the induction hypothesis (IH) that for all \( k < n \), \( n \in \mathbb{N}_0 \), the equality

\[
\text{fib}(k) = \left( \frac{1+\sqrt{5}}{2} \right)^k - \left( \frac{1-\sqrt{5}}{2} \right)^k
\]

holds, we can prove the premise underlying the implication of the principle of strong induction for all natural numbers \( n \) by investigating the following basic and inductive cases.
Proof of Theorem 6.2.2.1 (2)

Base case 1: \( n = 0 \). In this case, a straightforward calculation yields the desired equality:

\[
\text{fib}(0) = 0 = \frac{0}{\sqrt{5}} = \frac{1 - 1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}}
\]

Base case 2: \( n = 1 \). Again, a straightforward calculation yields as desired:

\[
\text{fib}(1) = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\frac{1}{2} + \frac{\sqrt{5}}{2} - \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}}
\]
Proof of Theorem 6.2.2.1 (3)

Inductive case: \( n \geq 2 \). Applying the IH for \( n-2, n-1 \) yields as desired:

\[
\begin{align*}
\text{fib}(n) &= \text{fib}(n-2) + \text{fib}(n-1) \\
(\text{2x IH}) &= \frac{(1+\sqrt{5})^{n-2} - (1-\sqrt{5})^{n-2}}{\sqrt{5}} + \frac{(1+\sqrt{5})^{n-1} - (1-\sqrt{5})^{n-1}}{\sqrt{5}} \\
&= \frac{\left[ (1+\sqrt{5})^{n-2} + (1+\sqrt{5})^{n-1} \right] - \left[ (1-\sqrt{5})^{n-2} + (1-\sqrt{5})^{n-1} \right]}{\sqrt{5}} \\
&= \frac{(1+\sqrt{5})^{n-2} \left[ 1 + \frac{1+\sqrt{5}}{2} \right] - (1-\sqrt{5})^{n-2} \left[ 1 + \frac{1-\sqrt{5}}{2} \right]}{\sqrt{5}} \\
&= \frac{(1+\sqrt{5})^{n-2} \left( \frac{1+\sqrt{5}}{2} \right)^2 - (1-\sqrt{5})^{n-2} \left( \frac{1-\sqrt{5}}{2} \right)^2}{\sqrt{5}} \\
(\ast) &= \frac{(1+\sqrt{5})^{n} - (1-\sqrt{5})^{n}}{\sqrt{5}} \\
&= \frac{(1+\sqrt{5})^{n} - (1-\sqrt{5})^{n}}{\sqrt{5}}
\end{align*}
\]
Proof of Theorem 6.2.2.1 (4)

The equality marked by (*) follows from the below two calculations that make use of the binomial formulae.

We have:

\[
\left(\frac{1 + \sqrt{5}}{2}\right)^2 = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2} = 1 + \frac{1 + \sqrt{5}}{2}
\]

Similarly we get:

\[
\left(\frac{1 - \sqrt{5}}{2}\right)^2 = \frac{1 - 2\sqrt{5} + 5}{4} = \frac{6 - 2\sqrt{5}}{4} = \frac{3 - \sqrt{5}}{2} = 1 + \frac{1 - \sqrt{5}}{2}
\]

□
Excursus: Which Rectangle looks ‘nicest’?

Rectangle 1

Rectangle 2

Rectangle 3
Most People say ‘Rectangle 3’!
Why?

\[ \frac{8 \text{ UoL}}{5 \text{ UoL}} = 1.6 \]
The value 1.6 comes close to

...the Golden Ratio:

\[ \phi = \frac{1 + \sqrt{5}}{2} = 1.61803398874989... \]

Intuitively:

- The ratio of section \( A \) and section \( B \) is the same as the ratio of section \( B \) and section \( C \)

\[ \frac{A}{B} = \frac{B}{C} \]

The value of this ratio is denoted by \( \phi \).
The Golden Ratio

...is perceived as harmonious:

\[
\frac{8 \text{ UoL}}{5 \text{ UoL}} = 1.6
\]
What is the value of $\phi$?

\[
\begin{array}{c|c|c}
\text{x} & \text{1}
\end{array}
\]

\[
\frac{x+1}{x} = \frac{x}{1} = \phi
\]

$\iff$  
\[
1 + \frac{1}{x} = x = \phi
\]

Thus:  
\[
1 + \frac{1}{\phi} = \phi
\]

$\implies$  
\[
\phi + 1 = \phi^2
\]

$\implies$  
\[
\phi^2 - \phi - 1 = 0
\]

$\implies$  
\[
\phi = \frac{1+\sqrt{5}}{2} = 1.618... \\
(\phi' = \frac{1-\sqrt{5}}{2} = -0.618...)
\]
The Golden Ratio

...not only in terms of the ratios of sections but also in terms of the ratios of the areas of e.g. rectangles:

\[
\begin{array}{c|c|c}
1 \text{ UoL} & (\phi - 1) \text{ UoL} \\
\hline
1^2 \text{ UoL}^2 = 1 \text{ UoL}^2 & 1^* (\phi - 1) \text{ UoL}^2 \\
& = (\phi - 1) \text{ UoL}^2
\end{array}
\]
The Golden Ratio, rectangles and the **Fibonacci numbers**:

\[
\begin{array}{c|c}
  21 & 13 \\
  \hline
  8 & 5
\end{array}
\]
The Golden Ratio and Fibonacci Numbers (2)

The sequence of Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ... 

The sequence of the ratios of the Fibonacci numbers:

\[
\begin{align*}
1/1 &= 1 \\
2/1 &= 2 \\
3/2 &= 1.5 \\
5/3 &= 1.6 \\
8/5 &= 1.6 \\
13/8 &= 1.625 \\
21/13 &= 1.615384615384615 \\
34/21 &= 1.619047619047619 \\
& \quad \vdots \\
1,346,269/832,040 &= 1.618033988750541
\end{align*}
\]
The Golden Ratio and Fibonacci Numbers (3)

...as the limit of the ratios of the Fibonacci numbers.

We have:

$$\lim_{n \to \infty} \frac{fib(n + 1)}{fib(n)} = \frac{1 + \sqrt{5}}{2} = \phi$$
Chapter 6.2.3

Structural Induction
Arithmetic Expressions

The set $\mathcal{AE}$ of (simple) arithmetic expressions is defined by the BNF rule:

$$e ::= n \mid v \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \star e_2) \mid (e_1/e_2)$$

where $n$ and $v$ stand for an (integer) numeral and variable, respectively.
Example A

Theorem (6.2.3.1)

Let $e \in \mathcal{AE}$, let $lp_e$ and $rp_e$ denote the number of left and right parentheses of $e$. Then we have:

$$lp_e = rp_e$$

Proof: By means of structural induction.
Proof of Theorem 6.2.3.1 (1)

Base case 1: Let $e \equiv n$, $n$ a numeral.

In this case $e$ does not contain any parentheses. This means $lp_e = 0 = rp_e$ which yields the desired equality of $lp_e$ and $rp_e$.

Base case 2: Let $e \equiv v$, $v$ a variable.

As above, we conclude $lp_e = 0 = rp_e$ obtaining the desired equality of $lp_e$ and $rp_e$ also in this case.
Proof of Theorem 6.2.3.1 (2)

Inductive case: Let \( e_1, e_2 \in \mathcal{AE} \), let \( \circ \in \{+, -, \ast, /\} \), and let
\[ e \equiv (e_1 \circ e_2). \]

By means of the induction hypothesis (IH) we can assume \( lp_{e_1} = rp_{e_1} \) and \( lp_{e_2} = rp_{e_2} \). This allows us to prove the desired equality of \( lp_e \) and \( rp_e \) thereby completing the proof as follows:

\[
\begin{align*}
(lp_{e_1} = rp_{e_1} \text{ and } lp_{e_2} = rp_{e_2}) & \implies (e \equiv (e_1 \circ e_2)) = lp_{e_1 \circ e_2} \\
& = 1 + lp_{e_1} + lp_{e_2} \\
& = rp_{e_1} + rp_{e_2} + 1 \\
& = rp_{e_1 \circ e_2} \\
(e \equiv e_1 \circ e_2) & = rp_e
\end{align*}
\]
Example B

Theorem (6.2.3.2)

Let $e \in \mathcal{AE}$, let $p_e$ and $op_e$ denote the number of parentheses and of operators of $e$, respectively. Then we have:

$$p_e = 2 \times op_e$$

Proof: By means of structural induction.
Proof of Theorem 6.2.3.2 (1)

Base case 1: Let \( e \equiv n \), \( n \) a numeral.

In this case \( e \) does not contain any parentheses or operators. This means \( p_e = 0 = op_e \), which yields as desired

\[
p_e = 0 = 2 \times 0 = 2 \times op_e
\]

Base case 2: Let \( e \equiv v \), \( v \) a variable.

As above, we conclude \( p_e = 0 = op_e \) obtaining the desired equality

\[
p_e = 0 = 2 \times 0 = 2 \times op_e
\]

in this case, too.
Proof of Theorem 6.2.3.2 (2)

Inductive case: Let $e_1, e_2 \in \mathcal{AE}$, let $\circ \in \{+, -, *, /\}$, and let $e \equiv (e_1 \circ e_2)$.

By means of the induction hypothesis (IH) we can assume that $p_{e_1} = 2 \times op_{e_1}$ and $p_{e_2} = 2 \times op_{e_2}$. With these equalities we obtain as desired:

\[
\begin{align*}
\left( e \equiv (e_1 \circ e_2) \right) &= p(e_1 \circ e_2) \\
&= 1 + p_{e_1} + p_{e_2} + 1 \\
&= 2 \times op_{e_1} + 2 + 2 \times op_{e_2} \\
&= 2 \times op_{e_1} + 2 \times 1 + 2 \times op_{e_2} \\
&= 2 \times (op_{e_1} + 1 + op_{e_2}) \\
&= 2 \times op(e_1 \circ e_2) \\
&= 2 \times op_e
\end{align*}
\]

\[\square\]
Example C

Theorem (6.2.3.3)

Let $e \in \mathcal{AE}$ be an arithmetic expression of depth $n$, let $\text{opd}_e$ denote the number of operands of $e$. Then we have:

$$\text{opd}_e \leq 2^n$$

Proof: By means of structural induction.
Proof of Theorem 6.2.3.3 (1)

Base case 1: Let $e \equiv n$, $n$ a numeral.

In this case $e$ has depth 0 and contains 1 operand. This yields as desired:

$$\text{opd}_e = \text{opd}_n = 1 = 2^0 \leq 2^0$$

Base case 2: Let $e \equiv v$, $v$ a variable.

As in the previous case $e$ has depth 0 and contains 1 operand. Again we obtain as desired:

$$\text{opd}_e = \text{opd}_v = 1 = 2^0 \leq 2^0$$
Proof of Theorem 6.2.3.3 (2)

Inductive case: Let $e_1, e_2 \in \mathcal{AE}$ be arithmetic expressions of depth $n$ and $m$, respectively. Without losing generality let $m \leq n$. Let $\circ \in \{+, -, \times, /\}$, and let $e \equiv (e_1 \circ e_2)$.

In this case expression $e$ has depth $n + 1$. By means of the induction hypothesis (IH) we can assume $\text{opd}_{e_1} \leq 2^n$ and $\text{opd}_{e_2} \leq 2^m$. Using these inequalities the proof can be completed as follows:

\[
\begin{align*}
\text{opd}_e & \quad (e \equiv (e_1 \circ e_2)) \\
& = \text{opd}_{(e_1 \circ e_2)} \\
& = \text{opd}_{e_1} + \text{opd}_{e_2} \\
& \leq 2^n + 2^m \quad (2x \text{ IH}) \\
& \leq 2^n + 2^n \quad (m \leq n) \\
& = 2 \times 2^n \\
& = 2^{n+1}
\end{align*}
\]

$\Box$
Chapter 6.3

Inductive Proofs on Algebraic Data Types
Chapter 6.3.1

Induction and Recursion
Induction and Recursion

...are closely related.

Intuitively:

▶ **Induction** describes things starting from something very simple, and building up from there: It is a **bottom-up** principle.

▶ **Recursion** starts from the whole thing, working backward to the simple case(s): It is a **top-down** principle.

Thus:

▶ **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 (e.g., defining **data structures**) resp. vice versa in others (e.g., defining **algorithms**) is often mostly due to historical reasons.

**Data types:**

```hs
data Tree = Leaf Integer
           | Node Tree Tree
```

**Algorithms:**

```hs
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n-1)
```
Examples

- **Inductive definition of (simple) arithmetic expressions:**
  1. Each numeral \( n \) and variable \( v \) is an (elementary) arithmetic expression.
  2. If \( e_1 \) and \( e_2 \) are arithmetic expressions, then also \((e_1 + e_2), (e_1 - e_2), (e_1 \times e_2), \) and \((e_1/e_2)\).
  3. Every arithmetic expression is inductively constructed by means of rules (r1) and (r2).

- **Recursive definition of merge sort:**
  A list of integers \( l \) is sorted by the following 3 steps:
  1. Split \( l \) into two sublists \( l_1 \) and \( l_2 \).
  2. Sort the sublists \( l_1 \) and \( l_2 \) recursively obtaining the sorted sublists \( sl_1 \) and \( sl_2 \).
  3. Merge the sorted sublists \( sl_1 \) and \( sl_2 \) into the sorted list \( sl \) of \( l \).
Summing up

- **Definitions of 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 **tree** is either empty or consists of a node and a set of subtrees.
  - 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 6.3.2
Inductive Proofs on Trees
Inductive Proofs on Trees

Let

\[ \text{data Tree} = \text{Leaf Integer} | \text{Node Tree Tree} \]

Theorem (6.3.2.1)

Let \( t \) be a value, i.e., a tree, of type \( \text{Tree} \) of depth \( n \), let \( \text{leaves}(t) \) denote the number of leafs of \( t \). Then we have:

\[ \text{leaves}(t) \leq 2^n \]

Proof: By means of structural induction.
Proof of Theorem 6.3.2.1 (1)

Base case: Let $t \equiv \text{Leaf } k$ for some integer $k$.

In this case $t$ has depth 0 and contains 1 leaf. This yields as desired:

$$\text{leaves}(t) = \text{leaves}(\text{Leaf } k) = 1 = 2^0 \leq 2^0$$
Proof of Theorem 6.3.2.1 (2)

Inductive case: Let $t_1$ and $t_2$ be two values of type $\text{Tree}$ of depth $n$ and $m$, respectively. Without losing generality let $m \leq n$, and let $t \equiv \text{Node } t_1 \ t_2$.

In this case $t$ is a tree of depth $n + 1$. By means of the inductive hypothesis (IH) we can assume $\text{leaves}(t_1) \leq 2^n$ and $\text{leaves}(t_2) \leq 2^m$. Using these inequalities the proof can be completed as follows:

$$\text{leaves}(t) = \text{leaves}(\text{Node } t_1 \ t_2) = \text{leaves}(t_1) + \text{leaves}(t_2)$$

$$(2 \times \text{IH}) \leq 2^n + 2^m$$

$$(m \leq n) \leq 2^n + 2^n = 2 \times 2^n = 2^{n+1}$$

$\square$
Chapter 6.3.3
Inductive Proofs on Lists
Preliminaries

Recall:

- A list is by definition finite.

Given a list, it is called

- partial, if it is built from the undefined list
- defined, if it is not partial and all its elements are defined

Note:

- For inductively proving properties on lists we have to distinguish the two cases of
  - defined lists (cf. Chapter 6.3.3)
  - partial lists with possibly undefined elements (cf. Chapter 6.3.4)
Inductive Proofs on Defined Lists

The inductive proof pattern for defined lists:

Let \( P \) be a property on lists.

1. **Base case:** Prove that \( P \) is true for the empty list, i.e. prove \( P([]) \).

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**.
- A property \( P \) proved using this pattern is true for lists with only **defined** elements of any **finite** length.
Example A: Induction on Lists (1)

Let

\[
\begin{align*}
\text{length} &: [a] \rightarrow \text{Int} \\
\text{length} [{}] &= 0 \\
\text{length} (_{:}\text{xs}) &= 1 + \text{length} \text{xs}
\end{align*}
\]

Lemma (6.3.3.1)

For all defined lists \( \text{xs}，\ \text{ys} \) holds:

\[
\text{length} (\text{xs} ++ \text{ys}) = \text{length} \text{xs} + \text{length} \text{ys}
\]

Proof by induction on the structure of \( \text{xs} \).
Example A: Induction on Lists (2)

Base case:

\[
\text{length}([ ] + + ys) \\
= \text{length} \; ys \\
= 0 + \text{length} \; ys \\
= \text{length} \; [ ] + \text{length} \; ys
\]

Inductive case:

\[
\text{length}((x : xs) + + ys) \\
= \text{length} \; (x : (xs + + ys)) \\
= 1 + \text{length} \; (xs + + ys) \\
\text{(IH)} \; = 1 + (\text{length} \; xs + \text{length} \; ys) \\
= (1 + \text{length} \; xs) + \text{length} \; ys \\
= \text{length} \; (x : xs) + \text{length} \; ys
\]

\[\square\]
Example B: Induction on Lists (1)

Let

\[
\text{listSum} :: \text{Num } a \Rightarrow [a] \rightarrow a \\
\text{listSum} \; [] = 0 \\
\text{listSum} \; (x:xs) = x + \text{listSum} \; xs
\]

Lemma (6.3.3.2)

For all defined lists \(xs\) holds:

\[
\text{listSum} \; xs = \text{foldr} \; (+) \; 0 \; xs
\]

Proof by induction on the structure of \(xs\).
Example B: Induction on Lists (2)

Base case:

\[
\text{listSum } [] \\
= 0 \\
= \text{foldr (+) 0 } []
\]

Inductive case:

\[
\text{listSum } (x : xs) \\
= x + \text{listSum } xs \\
(\text{IH}) = x + \text{foldr (+) 0 } xs \\
= \text{foldr (+) 0 } (x : xs)
\]
Example C: Induction on Lists w/ map (1)

Properties of \texttt{map} that can be proved by \textit{induction on lists}.

\[
\begin{align*}
\text{map} \ (f . g) & \quad = \quad \text{map} \ f \ . \ \text{map} \ g \\
\text{map} \ f . \ \text{tail} & \quad = \quad \text{tail} \ . \ \text{map} \ f \\
\text{map} \ f . \ \text{reverse} & \quad = \quad \text{reverse} \ . \ \text{map} \ f \\
\text{map} \ f . \ \text{concat} & \quad = \quad \text{concat} \ . \ \text{map} \ (\text{map} \ f) \\
\text{map} \ f \ ((xs++ys)) & \quad = \quad \text{map} \ f \ xs \ ++ \ \text{map} \ f \ ys \\
\text{map} \ (\lambda x \to x) & \quad = \quad \lambda y \to y \\
\text{Note:} \quad \lambda x \to x & \quad :: \quad a \to a \\
\lambda y \to y & \quad :: \quad [a] \to [a]
\end{align*}
\]
Lemma (6.3.3.3)

If $f$ is strict, it is true:

$$f \cdot \text{head} = \text{head} \cdot \text{map } f$$

**Proof** by induction on the structure of lists.
Example C: Induction on Lists w/ map (3)

Base case:  \[(f \ . \ head) \ []\]
\[
(\text{Def. of "."}) = f (head \ [])
\]
\[
= f \bot
\]
\[
(f \ \text{strict}) = \bot
\]
\[
= head \ []
\]
\[
(\text{Def. of map}) = head (map f \ [])
\]
\[
(\text{Def. of "."}) = (head . map f) \ []
\]

Inductive case:  \[f \ . \ head \ (x : xs)\]
\[
(\text{Def. of "."}) = f (head \ (x : xs))
\]
\[
= f \ x
\]
\[
= head (f \ x : \ map f \ xs)
\]
\[
(\text{Def. of map}) = head (map f \ (x : xs))
\]
\[
(\text{Def. of "."}) = (head . map f) \ (x : xs)
\]
\[\square\]
Example D: Induction on Lists w/ fold

Properties of \texttt{fold} that can be proved by induction on lists.

- If \texttt{op} is associative with \texttt{e \texttt{op} x = x} and \texttt{x \texttt{op} e = x} for all \texttt{x}, then for all finite \texttt{xs}
  \[ \texttt{foldr \ op \ e \ xs} = \texttt{foldl \ op \ e \ xs} \]
is true.

- If \texttt{x \texttt{op1} (y \texttt{op2} z)} = \texttt{(x \texttt{op1} y) \texttt{op2} z} and \texttt{x \texttt{op1} e} = \texttt{e \texttt{op2} x}, then for all finite \texttt{xs}
  \[ \texttt{foldr \ op1 \ e \ xs} = \texttt{foldl \ op2 \ e \ xs} \]
is true.

- For all finite \texttt{xs}
  \[ \texttt{foldr \ op \ e \ xs} = \texttt{foldl (flip \ op) \ e \ (reverse \ xs)} \]
is true.
Example D: Induction on Lists w/ (++)

Properties of (++) that can be proved by induction on lists.

- For all \(xs\), \(ys\), and \(zs\) it is true:
  \[
  (xs++ys) ++ zs = xs ++ (ys++zs)
  \]
  *(Associativity of (++))*

- \(xs ++ [] = [] ++ xs\)
  *([] is neutral element of (++))*
Example E: Induction on Lists w/ \textit{take}/\textit{drop}

Properties of \textit{take} and \textit{drop} that can be proved by \textit{induction} on lists.

- For all $m, n$ with $m, n \geq 0$ and finite $xs$ it is true:
  
  \begin{align*}
  \text{take } n \ xs + \text{drop } n \ xs &= xs \\
  \text{take } m \ . \ \text{take } n &= \text{take } (\text{min } m \ n) \\
  \text{drop } m \ . \ \text{drop } n &= \text{drop } (m+n) \\
  \text{take } m \ . \ \text{drop } n &= \text{drop } n \ . \ \text{take } (m+n)
  \end{align*}

- If $n \geq m$, it is additionally true:
  
  \text{drop } m \ . \ \text{take } n = \text{take } (n-m) \ . \ \text{drop } m
Example F: Induction on Lists w/ reverse

Properties of reverse that can be proved by induction on lists.

- For all finite \( xs \) it is true:
  \[
  \text{head (reverse } xs) = \text{last } xs
  \]
  \[
  \text{last (reverse } xs) = \text{head } xs
  \]

- For all finite \( xs \) with only defined elements it is true:
  \[
  \text{reverse (reverse } xs) = xs
  \]
Chapter 6.3.4

Inductive Proofs on Partial Lists
Preliminaries

Computations that

- fail to terminate
- are faulty, i.e., produce an error

do not give a proper, i.e., a defined value.

The value of such computations is called the

- undefined value.

The undefined value is usually denoted by $\perp$ ("bottom").
Examples

The function

\[
\text{buggy\_fac} :: \text{Integer} \rightarrow \text{Integer} \\
\text{buggy\_fac} \ n = (n-1) \times \text{buggy\_fac} \ n \\
\text{buggy\_fac} \ 0 = 1
\]

...induces for every argument a non-terminating computation.

The function

\[
\text{buggy\_div} :: \text{Integer} \rightarrow \text{Integer} \\
\text{buggy\_div} \ n = \text{div} \ n \ 0
\]

...produces an error for each argument called with.
The Undefined Value in Haskell

The **undefined value** \( \perp \)

- is an element of every data type of Haskell representing the value of a
  - faulty or non-terminating computation.

\( \perp \) can be considered the “**least accurate**” approximation of (ordinary) values of the corresponding data type.

The definition

- **Polymorphic**
  ```haskell
  bottom :: a
  bottom = bottom
  ``
  
  **Concrete**
  ```haskell
  bottom :: Integer
  bottom = bottom
  ``

is the most simple expression (of arbitrary type) whose evaluation leads to a **non-terminating computation** with value \( \perp \).
The Undefined Value and Lists

The undefined value ⊥ may occur as

- an “ordinary” element of a list
- a list itself.

Example:

\[
\begin{align*}
\text{lst1} &= 2 : \text{bottom} : 5 : 7 : [] \\
\text{lst2} &= 2 : 3 : 5 : 7 : \text{bottom} \\
\text{lst3} &= 2 : \text{bottom} : 5 : 7 : \text{bottom} \\
\text{lst4} &= 2 : 3 : 5 : 7 : []
\end{align*}
\]

Note:

- The occurrence of \text{bottom} in \text{lst1} and the first occurrence of \text{bottom} in \text{lst3} are of type \text{Integer}.
- The occurrence of \text{bottom} in \text{lst2} and the second occurrence of \text{bottom} in \text{lst3} are of type \text{[Integer]}. 
Defined and Partial Lists

Definition (6.3.4.1, Defined Values)
A value of a data type is defined, if it is not equal to ⊥.

Definition (6.3.4.2, Defined and Partial Lists)
A list is

- defined, if it is a list of defined values
- partial, if it is built from the undefined list, i.e., if its tail is the undefined list ⊥

Example:

- lst4 is a defined list, while lst1, lst2, lst3 are not.
- lst2 and lst3 are partial, while lst1 is neither defined nor partial (note: lst1 contains an undefined element but is not built from the undefined list).
Examples of Partial Lists

Successively increasingly defined partial lists:

- `bottom` (*the undefined list, i.e., the “least defined” partial list*)
- `1 : bottom` (*partial list*)
- `1 : 2 : bottom` (*partial list*)
- `1 : 2 : 3 : bottom` (*partial list*)
- `...`
- `1 : 2 : 3 : 4 : 5 : 6 : 7 : bottom` (*partial list*)
- `...`
Properties of Functions on Partial Lists (1)

\[
\begin{align*}
\text{reverse (lst1)} & \rightarrow [7,5 \ldots \text{followed by an infinite wait} \\
\text{reverse (lst2)} & \rightarrow \ldots\text{infinite wait} \\
\text{reverse (lst3)} & \rightarrow \ldots\text{infinite wait} \\
\text{reverse (lst4)} & \rightarrow [7,5,3,2]
\end{align*}
\]

\[
\begin{align*}
\text{head (tail (reverse lst1))} & \rightarrow 5 \\
\text{head (tail (tail (reverse lst1))]} & \rightarrow \ldots\text{infinite wait} \\
\text{last (lst1)} & \rightarrow 7
\end{align*}
\]

\[
\begin{align*}
\text{last (lst2)} & \rightarrow \ldots\text{infinite wait} \\
\text{head (tail (reverse lst2))} & \rightarrow \ldots\text{infinite wait}
\end{align*}
\]

\[
\begin{align*}
\text{head (reverse lst1)} & \rightarrow 7 \\
\text{head (tail (reverse lst1))} & \rightarrow 5 \\
\text{head (reverse (reverse lst1))} & \rightarrow 2 \\
\text{reverse (reverse (lst1))} & \rightarrow [2 \ldots \text{followed by an infinite wait}
\end{align*}
\]
Properties of Functions on Partial Lists (2)

length lst1 ->> 4
length lst2 ->> ...infinite wait
length lst3 ->> ...infinite wait
length lst4 ->> 4

length (take 3 lst1) ->> 3
length (take 2 lst2) ->> 2
length (take 3 lst3) ->> 3

length (take 4 lst4) ->> 4
length (take 5 lst4) ->> 4
length (take 4 lst2) ->> 4
length (take 5 lst2) ->> ...infinite wait
Properties of Functions on Partial Lists (3)

For understanding the different behaviours recall the definitions of \texttt{length} and \texttt{reverse}:

\begin{verbatim}
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs -- x is not evaluated!
\end{verbatim}

\begin{verbatim}
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse x ++ [x] -- x is evaluated!
\end{verbatim}

\begin{verbatim}
reverse :: [a] -> [a]
reverse = foldl (flip (:)) [] -- x is evaluated!
\end{verbatim}
Inductive Proofs on Lists Reconsidered

The inductive proof pattern introduced at the beginning of Chapter 6.3.3 holds for

- defined lists.

For inductive proofs of properties on partial lists (such as \texttt{lst2}) with possibly undefined elements (such as \texttt{lst3}) it has to be replaced by the inductive proof principle shown next.
Inductive Proofs on Partial Lists w/ Possibly Undefined Elements

Inductive proof pattern for partial lists with possibly undefined elements:

Let $P$ be a property on lists.

1. **Base case:** Prove that $P$ is true for the empty list and for the undefined list, i.e. prove $P([])$ and $P(⊥)$.

2. **Inductive case:** Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x : xs)$ is true, for $x$ being a defined and an undefined value (induction step).
Chapter 6.3.5
Inductive Proofs on Streams
Approximating Lists and Streams

Lists and Streams

- can be approximated by sequences of increasingly more accurate partial lists, also called approximants.
Approximating Lists by Partial Lists

The list

\[ [1,2,3,4,5] = 1 : 2 : 3 : 4 : 5 : [] \]

is approximated by the below sequence of partial lists that are increasingly more accurate approximations and ultimately culminate in the list \([1,2,3,4,5]\):

- bottom
- 1 : bottom
- 1 : 2 : bottom
- 1 : 2 : 3 : bottom
- 1 : 2 : 3 : 4 : bottom
- 1 : 2 : 3 : 4 : 5 : bottom
- 1 : 2 : 3 : 4 : 5 : []
Approximating Streams by Partial Lists (1)

The stream

\[1,2,3,4,5..\]

of natural numbers is the limit of the infinite sequence of increasing approximations of partial lists:

bottom
1 : bottom
1 : 2 : bottom
1 : 2 : 3 : bottom
1 : 2 : 3 : 4 : bottom
1 : 2 : 3 : 4 : 5 : bottom
... 
...
Note:

- Considering **partial lists approximations of streams** reminds to the strategy of partially outputting/printing streams by hitting `Ctrl-C` after some period of time.

- Extending this period of time further and further yields successively **more accurate approximations** of the stream.
Equality of Lists and Streams

**Definition (6.3.5.1, Equality of Lists)**

Two lists \(xs\) and \(ys\) are equal, if all their approximants are equal, i.e., if for all natural numbers \(n\), \(\text{take } n \ xs = \text{take } n \ ys\).

**Definition (6.3.5.2, Infinite Lists, Streams)**

A list \(xs\) is infinite or a stream, if for all natural numbers \(n\), \(\text{take } n \ xs \neq \text{take } (n+1) \ xs\).

**Definition (6.3.5.3, Equality of Streams)**

Two streams \(xs\) and \(ys\) are equal, if for all natural numbers \(n\), \(xs!!n = ys!!n\).
Extending Properties from Lists to Streams

Properties on lists

- can be extensible to streams, e.g.,
  
  \[ \text{take } n \; \text{xs } ++ \text{ drop } n \; \text{xs } = \text{xs} \]

- but need not be extensible to streams, e.g.,
  
  \[ \text{reverse} \; (\text{reverse} \; \text{xs}) ) = \text{xs} \]

Similarly, properties that are true for every partial list of an approximating sequence of partial lists

- can be true for their limit

- but need not be true for their limit, e.g., “this list is partial”.
Hence

Proving properties on streams thus demands for tailored
  ▶ proof strategies
that avoid such anomalies and paradoxes.

Fortunately
  ▶ The restriction “expressed as an equation in Haskell” is
  sufficiently to ensure that a property that is true for every
  partial list of an approximating sequence is also true for
  its limit.
Inductive Proofs on Streams

Inductive proof pattern for streams with only defined elements:
Let $P$ be a property on streams expressed as an equation in Haskell.

1. **Base case:** Prove that $P$ holds for the least defined list, i.e. prove $P(\bot)$ (instead of $P([])$).

2. **Inductive case:** Assume that $P(xs)$ is true (induction hypothesis) and prove that $P(x : xs)$ is true (induction step).
Example A: Induction on Streams (1)

Lemma (6.3.5.4)

For all streams $xs$ is true:

$$take \ n \ xs \ ++ \ drop \ n \ xs = xs$$

Proof by induction on the structure of $xs$. 
Example A: Induction on Streams (2)

Base case:

\[
\begin{align*}
take \; n \; \bot & \triangleright \triangleright \; drop \; n \; \bot \\
& = \; \bot \; \triangleright \triangleright \; drop \; n \; \bot \\
& = \; \bot
\end{align*}
\]

Inductive case:

\[
\begin{align*}
take \; n \; (x : xs) & \triangleright \triangleright \; drop \; n \; (x : xs) \\
& = \; x \; : \; (take \; (n - 1) \; xs \; \triangleright \triangleright \; drop \; (n - 1) \; xs) \\
(IH) & = \; x \; : \; xs
\end{align*}
\]
Chapter 6.4
Approximation
Proof by Approximation

...is an important principle

- for proving properties of infinite objects, e.g. equality of streams
- has been applied in Chapter 6.3.5.
- is more general than the usage suggested there.

...will be considered in more detail in this chapter.
Preliminaries

Definition (6.4.1, Partially Ordered Set)
A relation $R$ on $M$ is called a partially ordered set (or partial order) iff $R$ is reflexive, transitive, and anti-symmetric.

Definition (6.4.2, Chain)
Let $(P, \sqsubseteq)$ be a partially ordered set. A subset $C \subseteq P$ is called a chain of $P$, if the elements of $C$ are totally ordered.

Remark
- Refer to Appendix to recall the meaning of terms if necessary.
Domains

Definition (6.4.3, Domain)
A set $D$ with a partial order $\sqsubseteq$ is called a domain, if
1. $D$ has a least element $\bot$
2. $\bigsqcup C$ exists for every chain $C$ in $D$

Example
- Let $\mathcal{P}({\text{IN}})$ denote the power set of $\text{IN}$. Then $(\mathcal{P}({\text{IN}}), \sqsubseteq)$ with $\sqsubseteq =_{df} \subseteq$ is a domain with least element $\emptyset$ and $\bigsqcup C = \bigcup C$ for every chain $C$ in $\mathcal{P}({\text{IN}})$.

Note
- A domain is a (chain) complete partial order (cf. Appendix)
- The relation $\sqsubseteq$ of a domain is also called approximation order.
Approximation Order for Lists and Streams

Definition (6.4.4, Approximation Order)

We define the following relation on lists and streams:

\[ \bot \sqsubseteq xs \]
\[ [ ] \sqsubseteq xs = df \quad xs = [ ] \]
\[ x : xs \sqsubseteq y : ys = df \quad x \sqsubseteq y \land xs \sqsubseteq ys \]

Lemma (6.4.5, Domain Property of List Types)

Let \( a \) be a type such that its values form a domain. Then the values of the data types \([ a]\) form under the approximation order of Definition 6.4.4 a domain.
Approximating Lists by Partial Lists

By means of Definition 6.4.4, we have:

\[\bot \sqsubseteq x_0 : \bot \sqsubseteq x_0 : x_1 : \bot \sqsubseteq x_0 : x_1 : \ldots : x_n : \bot \]
\[\sqsubseteq x_0 : x_1 : \ldots : x_n : []\]

This finite set of approximations is a chain. We have:

\[\bigsqcup \{\bot, x_0 : \bot, x_0 : x_1 : \bot, x_0 : x_1 : \ldots : x_n : \bot, x_0 : x_1 : \ldots : x_n : []\} = x_0 : x_1 : \ldots : x_n : []\]
Approximating Streams by Partial Lists

Similarly, streams can be approximated by partial lists, too:

\[ \bot \sqsubseteq x_0 : \bot \sqsubseteq x_0 : x_1 : \bot \sqsubseteq x_0 : x_1 : \ldots : x_n : \bot \]
\[ \sqsubseteq x_0 : x_1 : \ldots : x_n : x_{n+1} : \bot \sqsubseteq \ldots \]

This infinite set of approximations is a chain. We have:

\[ \bigcup \{ \bot, x_0 : \bot, x_0 : x_1 : \bot, x_0 : x_1 : x_2 : \bot, \ldots \} = xs \]
Computing Partial Approximations

The function `approx` gives approximations of any list, stream:

\[
\text{approx} :: \text{Integer} \rightarrow [	ext{a}] \rightarrow [	ext{a}]
\]
\[
\text{approx} \ (n+1) \ [] \ = \ []
\]
\[
\text{approx} \ (n+1) \ (x:xs) \ = \ x \ : \ \text{approx} \ n \ x\ s
\]

Note:

- \(n+1\) matches only positive integers.
- Calling `approx n xs` with \(n\) smaller or equal to the length of \(xs\) will cause an error after generating the first \(n\) elements of the list, i.e., it generates the partial list

\[
x_0 : x_1 : \ldots : x_{n-1} : \bot
\]

- If \(n\) is greater than the length of \(xs\), the call `approx n xs` generates the whole list \(xs\).
Proof by Approximation

Lemma (6.4.6, Approximation)

For any list, stream \( xs \) holds:

\[
\bigcup_{n=0}^{\infty} \text{approx } n \; xs \; = \; xs
\]

Theorem (6.4.7, Approximation)

For any two lists, streams \( xs, \; ys \) hold:

\[
xs \; = \; ys \; \iff \; \forall \; n \; \in \; \mathbb{N}. \; \text{approx } n \; xs \; = \; \text{approx } n \; ys
\]
Note:

- The Approximation Theorem 6.4.7 is an important means for proving properties of streams.
- The inductive proof principle for streams of Chapter 6.3.5 is justified by Theorem 6.4.7.
Chapter 6.4: Further Reading

Chapter 6.5

Coinduction
Proof by Coinduction

...is another important principle

- for proving properties of infinite objects, e.g. equality of streams
- complements the principle of proof by approximation for proving properties of infinite objects (cf. Chapter 6.3.5)
- extends our tool box for proving properties of infinite objects like streams
Essence of Proof by Coinduction (1)

Proof by coinduction of equality of two infinite objects

- amounts to proving that the two objects exhibit the same observational behaviour.

For example, proving the equality of two streams $xs$ and $ys$ using the principle of proof by coinduction amounts to proving that

- $xs$ and $ys$ have the same heads
- the tails of $xs$ and $ys$ have the same observational behaviour
Essence of Proof by Coinduction (2)

Technically, proof by coinduction of the equality of two infinite objects $xs$ and $ys$ boils down to

- defining a bisimulation relation on $xs$ and $ys$, and proving them to be bisimilar.

Formalizing this requires the notions of a labeled transition system and a bisimulation relation.
Labeled Transition Systems

Definition (6.5.1, Labeled Transition System)

A labeled transition system is a triple \((Q, A, T)\) consisting of

- a set of states \(Q\)
- a set of action labels \(A\)
- a ternary relation \(T \subseteq Q \times A \times Q\), the transition relation.

Note:

- If \((q, a, p) \in T\), we write this as \(q \xrightarrow{a} p\).
Bisimulations

Definition (6.5.2, (Greatest) Bisimulation)

Let \((Q, A, T)\) be a labeled transition system. A bisimulation on \((Q, A, T)\) is a binary relation \(R\) on \(Q\) with the following properties.

If \(q R p\) and \(a \in A\) then

- If \(q \xrightarrow{a} q'\) then there is a \(p' \in Q\) with \(p \xrightarrow{a} p'\) and \(q' R p'\)
- If \(p \xrightarrow{a} p'\) then there is a \(q' \in Q\) with \(q \xrightarrow{a} q'\) and \(q' R p'\)

We denote the greatest bisimulation on \(Q\) by \(\sim\).
Example

Consider the following decimal representations of $\frac{1}{7}$

- $0.\overline{142857}$
- $0.\overline{1428571}$
- $0.\overline{14285714}$
- $0.\overline{142857142857142}$

and the relation $R$ ‘having the same infinite expansion’ on decimal representations.

Then

- $R$ is a bisimulation on decimal representations
- $0.\overline{142857}, 0.\overline{1428571}, 0.\overline{14285714}, 0.\overline{142857142857142}$ are all bisimilar.
Illustration
Bisimilar

Definition (6.5.3, Bisimilar)

Let \((Q, A, T)\) be a labeled transition system, and let \(p, q \in Q\).

Then \(p\) and \(q\) are called bisimilar, if they are related by a bisimulation on \(Q\).
Proof by Coinduction (1)

The general pattern of a proof by coinduction for proving the equality of infinite objects:

Let \( x \) and \( y \) be two infinite objects.

To prove that \( x \) and \( y \) are equal, show that they exhibit the same behaviour, i.e. prove that \( x \sim y \):

\[ a \sim b \iff \exists R. \ (R \text{ is a bisimulation, and } a R b) \]
A proof matching the preceding pattern is called a proof by coinduction.

Next, we are going to show how to use this pattern to prove equality of streams.
Proof by Coinduction (3)

To this end, we introduce the following notation:

If $f = [f_0, f_1, f_3, f_4, f_5, \ldots]$ is a stream, then $f_0$ denotes the head and $\bar{f}$ the tail of $f$, i.e., $f = f_0 : \bar{f}$.

Note:
- A stream $f$ can be considered a labeled transition system.

Illustration

Stream $f$ as a labeled transition system
Equality of Streams

Let \( f = [f_0, f_1, f_3, f_4, f_5, \ldots] \) and \( g = [g_0, g_1, g_3, g_4, g_5, \ldots] \) be two streams.

Then

- \( f \) and \( g \) are equal iff they exhibit the same behaviour
  iff \( \forall i \in \mathbb{N}_0. f_i = g_i \)

This boils down to

- \( f \) and \( g \) are equal iff \( f \sim g \), i.e., \( f_0 = g_0 \) and \( \bar{f} \sim \bar{g} \)
  with \( f \xrightarrow{f_0} \bar{f} \) and \( g \xrightarrow{g_0} \bar{g} \).
Stream Bisimulation

Definition (6.5.4, Stream Bisimulation)

A stream bisimulation on a set $A$ is a relation $R$ on $[A]$ with the following property.

If $f, g \in [A]$ and $f R g$ then both $f_0 = g_0$ and $\bar{f} R \bar{g}$.

Illustration

Bisimulation between two streams $f$ and $g$
Proof by Coinduction w/ Stream Bisimulations

The general pattern of a proof by coinduction using stream bisimulations of $f \sim g$, where $f, g \in [A]$:

1. Define a relation $R$ on $[A]$
2. Prove that $R$ is a stream bisimulation, with $f R g$. 


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

...another important proof principle.

Fixed point induction allows proving properties of functions on ordered sets, such as complete partial orders, lattices, and the like (cf. Appendix).
Admissible Predicates

**Definition (6.5.1, Admissible Predicate)**

Let \((C, \sqsubseteq)\) be a complete partial order (CPO), and let \(\psi : C \rightarrow IB\) be a predicate on \(C\).

The predicate \(\psi\) is called **admissible** iff for every chain \(D \subseteq C\) holds:

\[
\text{if } \psi(d) = \text{true} \text{ for all } d \in D \text{ then } \psi(\bigsqcup D) = \text{true}
\]
Fixed Point Induction

The general pattern of a proof by fixed point induction:

**Theorem (6.5.2, 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 $C$.

If for all $c \in C$ holds that

$$\psi(c) = true \text{ implies } \psi(f(c))$$

then

$$\psi(\mu f) = true$$

where $\mu f$ denotes the least fixed point of $f$. 
Note

Streams

▶ form a domain resp. CPO (cf. Chapter 6.4 and Appendix)

Hence, fixed point induction is a relevant proof technique for a functional programmer.
Chapter 6.6: Further Reading


Chapter 6.7
Other Approaches, Verification 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 based on induction and equality reasoning 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 with 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 the type inference algorithm as an 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 match failures; evaluated on ‘real’ programs.

- ...
Chapter 6: Further Reading (1)


Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18, Programme verifizieren und testen)

Chapter 6: Further Reading (2)


Chapter 6: Further Reading (3)


Chapter 6: Further Reading (4)


Chapter 6: Further Reading (6)


Chapter 6: Further Reading (7)


Chapter 6: Further Reading (8)


Chapter 6: Further Reading (9)


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 (10)


Part IV

Advanced Language Concepts
Chapter 7

Functional Arrays
Imperative Arrays

For imperative arrays holds:

- A value of the 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.
Lists and Functional Arrays

(Functional) lists

- do not enjoy the favorable list of characteristics of imperative arrays; most importantly, values of a list cannot be accessed or updated in constant time.
  - Using \((!!)\) to access the \(i\)th element of a list takes a number of steps proportional to \(i\).

- Lists can be arbitrarily long, potentially even infinite.

Functional arrays

- are designed and implemented to get as close as possible to the characteristics of imperative arrays.
  - Using \((!)\) to access the \(i\)th element of an array takes a constant number of steps, regardless of \(i\).

- Arrays are of a fixed size which must be defined at the time the array is (first) created.
Functional Arrays

Functional arrays

- are not part of the standard prelude `Prelude.hs` of Haskell.

Various libraries

- provide different kinds of functional arrays
  - import `Array`
  - import `Data.Array.IArray`
  - import `Data.Array.Diff`

Important variants of functional arrays

- **Static** arrays (w/out destructive update)
- **Dynamic** arrays (w/ destructive update)
Creating static arrays

import Array

There are three functions for creating static arrays:

- `array bounds list_of_associations`
- `listArray bounds list_of_values`
- `accumArray f init bounds list_of_associations`
Creating Static Arrays

The three functions for creating static arrays in more detail:

- `array :: Ix a => (a,a) -> [(a,b)] -> Array a b
  array bounds list_of_associations`

- `listArray ::(Ix a) => (a,a) -> [b] -> Array a b
  listArray bounds list_of_values`

- `accumArray :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)] -> Array a b
  accumArray f init bounds list_of_associations`
The Type Class Ix

Ix denotes the class of types that are (mainly) used for indices of arrays.

- Members of the type class Ix must provide implementations of the functions
  - range
  - index
  - inRange
  - rangeSize

- Ix inherits from the type class Ord (and indirectly from the type class Eq):

```
class (Ord a) => Ix a where
  range    :: (a,a) -> [a]
  index    :: (a,a) -> a -> Int
  inRange  :: (a,a) -> a -> Bool
  rangeSize :: (a,a) -> Int
```
Creating Static Arrays: The 1st Mechanism

The first and most fundamental array creation mechanism:

- \( \text{array} : \text{Ix } a \Rightarrow (a, a) \rightarrow [(a, b)] \rightarrow \text{Array } a \text{ b} \)

\[ \text{array bounds list_of_associations} \]

Meaning of the arguments:

- **bounds**: gives the value of the lowest and the highest index in the array.

**Example**: **bounds** of a

- zero-origin vector of five elements: \((0, 4)\)
- one-origin 10 by 10 matrix: \(((1, 1), (10, 10))\)

**Note**: The values of the bounds can be arbitrary expressions.

- **list_of_associations**: a list of associations, where an association is of the form \((i, x)\) meaning that the value of the array element \(i\) is \(x\).
Examples

The expressions

\[ a' = \text{array}\ (1,4)\ [(3,\ 'c'),(2,\ 'a'),(1,\ 'f'),(4,\ 'e')] \]
\[ f \ n = \text{array}\ (0,n)\ [(i,i*i)\ |\ i <- [0..n]] \]
\[ m = \text{array}\ ((1,1),(2,3)) \]
\[ \qquad [(i,j),(i*j))\ |\ i<-[1..2],\ j<-[1..3]] \]

have type

\[ a' :: \text{Array}\ \text{Int}\ \text{Char} \]
\[ f :: \text{Int} \to \text{Array}\ \text{Int}\ \text{Int} \]
\[ m :: \text{Array}\ (\text{Int},\text{Int})\ \text{Int} \]

and value

\[ a' \to> \text{array}\ (1,4)\ [(1,\ 'f'),(2,\ 'a'),(3,\ 'c'),(4,\ 'e')] \]
\[ f 3 \to> \text{array}\ (0,3)\ [(0,0),(1,1),(2,4),(3,9)] \]
\[ m \to> \text{array}\ ((1,1),(2,3))\ [((1,1),1),((1,2),2),\]
\[ \qquad ((1,3),3),((2,1),2),\]
\[ \qquad ((2,2),4),((2,3),6)] \]
Properties of Array Creation

In general:

Arrays have type

- Array a b where
  - a: represents the type of the index
  - b: represents the type of the value

Moreover:

- 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: array is strict in the bounds but non-strict (lazy) in the values. In particular, an array can thus contain ‘undefined’ elements.
Example

The computation of the Fibonacci numbers:

\[
\text{fibs } n = a \\
\text{where } a = \text{array} \ (1,n) \ ([(1,0), (2,1)] ++ \ ((i, a!(i-1) + a!(i-2)) \ | \ i \ <- \ [3..n]])
\]

Applications:

\[
\text{fibs 3 } \rightarrow \rightarrow \ \text{array} \ (1,3) \ [(1,0),(2,1),(3,1)]
\]
\[
\text{fibs 5 } \rightarrow \rightarrow \ \text{array} \ (1,5) \ [(1,0),(2,1),(3,1), (4,2),(5,3)]
\]
\[
\text{fibs 10 } \rightarrow \rightarrow \ \text{array} \ (1,10) \ [(1,0),(2,1),(3,1), (4,2),(5,3),(6,5), (7,8),(8,13),(9,21), (10,34)]
\]
Example (Cont’d)

More Applications:

```
fibs 5!5        -->  3
fibs 10!10      -->  34

fibs 100!10     -->  34  -- Thanks to lazy evaluation
                        -- 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
```
The Array Access Function (!)

The signature of the array access function (!):

(!) :: Ix a => Array a b -> a -> b

Recall: The index type must be an element of type class \texttt{Ix}, which defines operations specifically needed for index computations.
Example (Cont’d)

Note:

▶ The declaration of a in a where-clause is crucial for performance.
▶ The local declaration of a avoids creating new arrays during computation.

For comparison consider:

\[
\begin{align*}
a\ n &= \text{array (1,n) } \begin{bmatrix} (1,0), & (2,1) \end{bmatrix} \ + \\
& \quad \left. \begin{bmatrix} (i, a\ n!(i-1) + a\ n!(i-2)) \right| i \leftarrow [3..n] \right) \\
xfibs\ n &= a\ n
\end{align*}
\]
Example (Cont’d)

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 10 ->> array (1,10) [(1,0),(2,1),(3,1),(4,2),
                           (5,3),(6,5),(7,8),(8,13),
                           (9,21),(10,34)]

xfibs 5!5  ->> 3
xfibs 10!10 ->> 34
xfibs 25!20 ->> 4.181
xfibs 25!25 ->> ...takes too long to be feasible!

Note: Though correct, the evaluation of xfibs n is most inefficient due to the generation of new arrays during computation.
Creating Static Arrays: The 2nd Mechanism

The second array creation mechanism:

- \( \text{listArray} :: (\text{Ix } a) \rightarrow (a,a) \rightarrow \text{[b]} \rightarrow \text{Array } a \text{ b} \)

\[ \text{listArray } \text{bounds } \text{list_of_values} \]

Meaning of the arguments:

- \text{bounds}: gives the value of the lowest and the highest index in the array.
- \text{list_of_values}: a list of values.

The function \text{listArray}

- is useful for the frequently occurring case where an array is constructed from a list of values in index order.

Example:

\[ a'' = \text{listArray } (1,4) \text{ "face"} \]

\[ a'' \rightarrow \text{array } (1,4) [(1, 'f'), (2, 'a'), (3, 'c'), (4, 'e')] \]
Creating Static Arrays: The 3rd Mechanism

The third array creation mechanism:

\[
\text{accumArray :: (Ix a) => (b -> c -> b) -> b}\\
\rightarrow \text{((a,a) -> [(a,c)]]} \rightarrow \text{Array a b}
\]

\[\text{accumArray } f \text{ init bounds list_of_associations}\]

...removes the restriction that a given index may appear at most once in the association list. Instead, ‘conflicting’ indices are accumulated via a function \( f \).

Meaning of the arguments:

1. \( f \): an accumulation function.
2. \( \text{init} \): gives the (default) value the entries of the array shall be initialized with.
3. \( \text{bounds} \): gives the value of the lowest and the highest index in the array.
4. \( \text{list_of_associations} \): a list of associations.
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:

```
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
```

A Prime Number Test

...using the function \texttt{accumArray}:

\begin{verbatim}
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]]
\end{verbatim}

Applications:

(\texttt{primes 100})!1 \rightarrow \text{Program error: Ix.index: index out of range}

(\texttt{primes 100})!2 \rightarrow \text{True}

(\texttt{primes 100})!4 \rightarrow \text{False}

(\texttt{primes 100})!71 \rightarrow \text{True}

(\texttt{primes 100})!100 \rightarrow \text{False}

(\texttt{primes 100})!101 \rightarrow \text{Program error: Ix.index: index out of range}
A Prime Number Test (Cont’d)

More Applications:

```
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
```
Array Operators

Array operators are:

- `!`: array subscripting.
- `bounds`: yields bounds of an array.
- `indices`: yields list of indices of an array.
- `elems`: yields list of elements of an array.
- `assocs`: yields list of associations of an array.
- `//`: array updating – the operator `//` takes an array and a list of associations and returns a new array identical to the left argument except for every element specified by the right argument list.

This means: `//` does not perform a destructive update!

- ...
Array Operators (Cont’d)

- (!) :: (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
- ...
Illustrating the Usage of Array Operators

Let

\[
m = \text{array} \left( (1,1), (2,3) \right) \left[ \left( (i,j), (i \times j) \right) \mid i \leftarrow [1..2], j \leftarrow [1..3] \right] \]

Then

\[
m \rightarrow> \text{array} \left( (1,1), (2,3) \right) \left[ \left( (1,1), 1 \right), \left( (1,2), 2 \right), \left( (1,3), 3 \right), \left( (2,1), 2 \right), \left( (2,2), 4 \right), \left( (2,3), 6 \right) \right] \]

\[
m!(1,2) \rightarrow> 2, \ m!(2,2) \rightarrow> 4, \ m!(2,3) \rightarrow> 6
\]

\[
\text{bounds } m \rightarrow> ((1,1),(2,3))
\]

\[
\text{indices } m \rightarrow> [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
\]

\[
\text{elems } m \rightarrow> [1,2,3,2,4,6]
\]

\[
\text{assocs } m \rightarrow> [(1,1),1, (1,2),2, (1,3),3, (2,1),2, (2,2),4, (2,3),6]
\]

\[
m \ // \ [(1,1),4, (2,2),8)]
\rightarrow> \text{array} \left( (1,1), (2,3) \right) \left[ \left( (1,1), 4 \right), \left( (1,2), 2 \right), \left( (1,3), 3 \right), \left( (2,1), 2 \right), \left( (2,2), 8 \right), \left( (2,3), 6 \right) \right]
\]
Illustrating the Update Operator

The **histogram** function:

\[
\text{histogram} \ (\text{lower,upper}) \ \text{xs} \\
= \ \text{updHist} \ (\text{array} \ (\text{lower,upper}) \\
\quad \quad \quad \quad \quad \quad \ [(i,0) \mid i<-[\text{lower}..\text{upper}]]) \ \text{xs}
\]

\[
\text{updHist} \ a \ [] = a \\
\text{updHist} \ a \ (x:xs) = \text{updHist} \ (a \ // \ [(x, (a!x + 1))]) \ xs
\]

**Application:**

\[
\text{histogram} \ (0,9) \ [3,1,4,1,5,9,2] \\
\rightarrow \ \text{array} \ (0,9) \ [(0,0),(1,2),(2,1),(3,1),(4,1), \ (5,1),(6,0),(7,0),(8,0),(9,1)]
\]
Illustrating the `accum` Operator

Instead of replacing the old value, values with the same index could also be combined using the predefined:

```haskell
accum :: (Ix a) => (b -> c -> b) -> Array a b
      -> [(a,c)] -> Array a b
```

Application:

```haskell
accum (+) m [((1,1),4), ((2,2),8)]
```

```haskell
-> array ((1,1),(2,3))
```

```haskell
[(1,1),5), ((1,2),2), ((1,3),3),
  ((2,1),2), ((2,2),12), ((2,3),6)]
```

Note:

- The result is a new matrix identical to `m` except for the elements `(1,1)` and `(2,2)` to which 4 and 8 have been added, respectively.
Higher-Order Array Functions

Higher-order functions can be defined on arrays just as on lists.

For illustration consider:

- The expression
  
  \[
  \text{map } (\lambda x \to x \times 10) \ a
  \]

  ...creates a new array where all elements of \( a \) are multiplied by 10.

- The expression
  
  \[
  \text{ixmap } b \ f \ a = \text{array } b \ [(k, a ! f k) \mid k \leftarrow \text{range } b]
  \]

  ...with

  \[
  \text{ixmap :: (Ix } a, \text{Ix } b) \Rightarrow (a,a) \to (a \to b) \to \text{Array } b \ c \to \text{Array } a \ c
  \]
Higher-Order Array Functions (Cont’d)

The functions `row` and `col` return a row and a column of a matrix:

```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
```
Higher-Order Array Functions (Cont’d)

Applications:

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
Dynamic Arrays

Creating dynamic arrays

```haskell
import Data.Array.Diff
```

Instead of

- type `Array`

we now have to use

- type `DiffArray`

...everything else remains the same.
Summing up

Static Arrays

- **Access operator (!):** access to each array element in constant time.

- **Update operator (//):** no destructive updates; instead an identical copy of the argument array is created except of those elements which were ‘updated.’ Updates thus do not take constant time.

Dynamic Arrays

- **Update operator (//):** destructive updates; updates take constant time per index.

- **Access operator (!):** access to array elements may sometimes take longer as for static arrays.
Summing up (Cont’d)

Recommendation

- Dynamic arrays should only be used if constant time updates are crucial for the application.
- Often, updates can completely be avoided by smartly written recursive array constructions (cp. the prime number test in this chapter).
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
Concrete vs. Abstract Data Types (1)

Concrete Data Types (CDTs)

- A new CDT is specified by naming its values.
- With the exception of functions, each value of a type is described by a unique expression in terms of constructors.
- Using definition by pattern matching as a basis, these expressions can be generated, inspected, and modified in various ways.
- There is no need to specify the operations associated with a type.
- The Haskell means for defining CDTs are algebraic data type definitions.
Concrete vs. Abstract Data Types (2)

Abstract Data Types (ADTs)

- A new ADT is not specified by naming its values but by naming its operations.
- How values are represented is thus less important than what operations are provided for manipulating them, whose meaning, of course, has to be described
  - degree of freedom for the implementation!
  - Information hiding!
- There is no dedicated means in Haskell for defining ADTs; ADTs, however, can be defined using modules.
Concrete vs. Abstract Data Types (3)

Implementing an ADT

▶ When implementing an ADT, a representation of its values has to be provided, and a definition of the operations of the type in terms of this representation.
▶ The representation can be chosen e.g. for grounds of simplicity or efficiency.
▶ It has to be shown that the implemented operations satisfy the prescribed relationships.
In the following...

...we consider abstract data types for

- Stacks
- Queues
- Priority Queues
- Tables
Chapter 8.1

Stacks
The Abstract Data Type Stack (1)

The user-visible interface specification of the Abstract Data Type (ADT) Stack:

```haskell
module Stack (Stack, push, pop, top, emptyStack, stackEmpty) where

push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
emptyStack :: Stack a
stackEmpty :: Stack a -> Bool
```

Note: In a stack elements are removed in a last-in/first-out (LIFO) order.
The Abstract Data Type Stack (2)

A user-invisible implementation of Stack as an algebraic data type (using data):

```haskell
data Stack a = EmptyStk
               | Stk a (Stack a)

push x s = Stk x s

pop EmptyStk  = error "pop from an empty stack"
pop (Stk _ s) = s

top EmptyStk  = error "top from an empty stack"
top (Stk x _) = x

emptyStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _ = False
```
The Abstract Data Type Stack (3)

A user-invisible implementation of Stack as an algebraic data type (using newtype):

```haskell
newtype Stack a = Stk [a]

push x (Stk xs) = Stk (x:xs)

pop (Stk []) = error "pop from an empty stack"
pop (Stk (_:xs)) = Stk xs

top (Stk []) = error "top from an empty stack"
top (Stk (x:_)) = x

emptyStack = Stk []

stackEmpty (Stk []) = True
stackEmpty (Stk _) = False
```
Displaying Stacks (1)

Note:

- The constructors `EmptyStk` and `Stk` are not exported from the module.
- This implies that a user of the module can not use or create a `Stack` by any other way than the operations exported by the module.
- While this is actually so desired, the user can also not display a value of type `Stack` except for the crude and cumbersome way of completely popping the whole stack.

Next, we describe and compare two ways to display stacks and their elements more elegantly.
Displaying Stacks (2)

The easy way: Using a deriving-clause

data Stack a = EmptyStk
  | Stk a (Stack a) deriving Show

newtype Stack a = Stk [a] deriving Show

Effect:

push 3 (push 2 (push 1 emptyStack))
  --> Stk 3 (Stk 2 (Stk 1 EmptyStk))

push 3 (push 2 (push 1 emptyStack))
  --> Stk [3,2,1]
Displaying Stacks (3)

Using the `deriving`-clause for type class `Show`:

**Advantage**
- Simplicity, no effort.

**Disadvantage**
- The implementation of the ADT `Stack` is disclosed to the programmer (though the user cannot access the representation in any way outside the module definition of the ADT `Stack`).
Displaying Stacks (4)

A smarter solution:

```haskell
instance (Show a) => Show (Stack a) where
  showsPrec _ EmptyStk str = showChar '‐' str
  showsPrec _ (Stk x s) str
    = shows x (showChar '|' (shows s str))

instance (Show a) => Show (Stack a) where
  showsPrec _ (Stk []) str = showChar '‐' str
  showsPrec _ (Stk (x:xs)) str
    = shows x (showChar '|' (shows (Stk xs) str))
```

Effect:

```
push 3 (push 2 (push 1 emptyStack)) ->> 3|2|1|‐
```
Displaying Stacks (5)

This way:

- The implementation of the ADT Stack remains hidden. It is not disclosed to the user.
- The output is the same for both implementations!

Note:

- The first argument of `showsPrec` is an unused precedence value.
Last but not least

An implementation of stacks in terms of

▶ predefined lists in Haskell: \texttt{type Stack a = [a]}

would be possible, too.

Advantage

▶ Even less conceptual overhead as for the implementation in terms of \texttt{newtype Stack a = Stk [a]}

Disadvantage

▶ All predefined functions on lists would be available on stacks, too.

▶ Many of these, however, e.g. for reversing a list, for picking some arbitrary element, are not meaningful for stacks.

▶ Implementing stacks in terms of predefined lists would not automatically exclude the application of such meaningless functions but require to explicitly abstain from them. Conceptually, this is disadvantageous.
Chapter 8.2

Queues
The Abstract Data Type Queue (1)

The user-visible interface specification of the Abstract Data Type (ADT) Queue:

```haskell
module Queue (Queue, emptyQueue, queueEmpty, enQueue, deQueue, front) where

emptyQueue :: Queue a
queueEmpty :: Queue a -> Bool
enQueue :: a -> Queue a -> Queue a
deQueue :: Queue a -> Queue a
front :: Queue a -> a
```

Note: In a queue elements are removed in a first-in/first-out (FIFO) order.
The Abstract Data Type Queue (2)

A user-invisible implementation of Queue as an algebraic data type:

```haskell
newtype Queue a = Q [a]

emptyQueue = Q []

queueEmpty (Q []) = True
queueEmpty _ = False

enQueue x (Q q) = Q (q ++ [x])

deQueue (Q []) = error "deQueue: empty queue"
deQueue (Q (_:xs)) = Q xs

front (Q []) = error "front: empty queue"
front (Q (x:_)) = x
```
Displaying Queues

The easy way: Using a deriving-clause

\[
\text{newtype Queue } a = \text{Q } [a] \text{ deriving Show}
\]

Advantages, disadvantages:

Chapter 8.3
Priority Queues
The Abstract Data Type PQueue (1)

The user-visible interface specification of the Abstract Data Type (ADT) PQueue:

module PQueue (PQueue,emptyPQ,pqEmpty,enPQ,dePQ,frontPQ) where

emptyPQ :: PQueue a
pqEmpty :: PQueue a -> Bool
enPQ :: (Ord a) => a -> PQueue a -> PQueue a
dePQ :: (Ord a) => PQueue a -> PQueue a
frontPQ :: (Ord a) => PQueue a -> a

Note: In a priority queue each entry has a priority associated with it. The dequeue operation always removes the entry with the highest (or lowest) priority. Technically, this is ensured by the enqueue operation, which places a new element according to its priority in a queue.
The Abstract Data Type PQueue (2)

A user-invisible implementation of PQueue as an algebraic data type:

newtype PQueue a = PQ [a]

emptyPQ = PQ []

pqEmpty (PQ []) = True
pqEmpty _ = False

enPQ x (PQ q) = PQ (insert x q)
  where insert x [] = [x]
       insert x r@(e:r') | x <= e = x:r
       | otherwise = e:insert x r'

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

frontPQ (PQ []) = error "frontPQ: empty priority queue"
frontPQ (PQ (x:_)) = x
Displaying Priority Queues

The easy way: Using a deriving-clause

newtype PQueue a = PQ [a] deriving Show

Advantages, disadvantages:
Chapter 8.4
Tables
The Abstract Data Type Table (1)

The user-visible interface specification of the Abstract Data Type (ADT) Table:

module Table (Table,newTable,findTable,updTable)
where

newTable :: (Eq b) => [(b,a)] -> Table a b
findFirst :: (Eq b) => Table a b -> b -> a
updTable :: (Eq b) => (b,a) -> Table a b
           -> Table a b

Note:

- The function newTable takes a list of (index,value) pairs and returns the corresponding table.
- The functions findTable and updTable are used to retrieve and update values in the table.
The Abstract Data Type Table (2)

A user-invisible implementation of Table as a function:

```haskell
newtype Table a b = Tbl (b -> a)

newTable assocs =
  foldr updTable (Tbl (_ -> error "updTable: item not found")) assocs

findTable (Tbl f) i = f i

updTable (i,x) (Tbl f) = Tbl g
  where g j | j==i = x
            | otherwise = f j
```
Displaying Tables Represented as Functions

Using an instance-clause

```haskell
instance Show (Table a b) where
  showsPrec _ _ str = showString "<<A Table>>" str
```
The Abstract Data Type Table (3)

A user-invisible implementation of Table as a list:

```haskell
newtype Table a b = Tbl [(b,a)]
newTable t = Tbl t

findTable (Tbl []) i
  = error "findTable: item not found"
findTable (Tbl ((j,v):r)) i
  | i==j    = v
  | otherwise = findTable (Tbl r) i

updTable e (Tbl []) = Tbl [e]
updTable e@(i,_) (Tbl (e@(j,_):r))
  | i==j    = Tbl (e':r)
  | otherwise = Tbl (e:r')
where Tbl r' = updTable e' (Tbl r)
```
Displaying Tables Represented as Lists

The easy way: Using a deriving-clause

```haskell
newtype Table a b = Tbl [(b,a)] deriving Show
```

Advantages, disadvantages:

The Abstract Data Type Table (4)

The user-visible interface specification of the Abstract Data Type (ADT) Table for implementation as an Array:

```
module Table (Table,newTable,findTable,updTable)
where

newTable :: (Ix b) => [(b,a)] -> Table a b
findTable :: (Ix b) => Table a b -> b -> a
updTable :: (Ix b) => (b,a) -> Table a b
            -> Table a b
```

Note:
- The function `newTable` takes a list of `(index,value)` pairs and returns the corresponding table.
- The functions `findTable` and `updTable` are used to retrieve and update values in the table.
A user-invisible implementation of Table as an Array:

```haskell
newtype Table a b = Tbl (Array b a)

newTable l = Tbl (array (lo,hi) l)
  where indices = map fst l
          lo = minimum indices
          hi = maximum indices

findTable (Tbl a) i = a ! i

updTable p@(i,x) (Tbl a) = Tbl (a // [p])
```

The Abstract Data Type Table (5)
The Abstract Data Type Table (6)

Note:

- The function `newTable` determines the boundaries of the new table by computing the maximum and the minimum key in the association list.
- In the function `findTable`, access to an invalid key returns a system error, not a user error.
Displaying Tables Represented as Arrays

The easy way: Using a deriving-clause

newtype Table a b = Tbl (Array b a) deriving Show

Advantages, disadvantages:

Chapter 8.5

Summing Up
Summing up

Benefits of using abstract data types:

- **Information hiding**: Only the interface is publicly known; the implementation itself is hidden. This offers:
  - **Security** of the data (structure) from uncontrolled or unintended/not admitted access.
  - **Simple exchangeability** of the underlying implementation (e.g. simplicity vs. performance).
  - **Work-sharing** of implementation.

There are many more implementations of data types in terms of an abstract data type. E.g.:

- Sets
- Heaps
- (Binary Search) Trees
- Arrays
- ...
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 !, //
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: An Abstract Data Type (Cont’d)

accumArray :: (Ix a) => (b -> c -> b) -> b
            -> (a,a) -> [(a,c)] -> Array a b
(///)      :: (Ix a) => Array a b -> [(a,b)]
            -> Array a b
accum      :: (Ix a) => (b -> c -> b) -> Array a b
            -> [(a,c)] -> Array a b
ixmap      :: (Ix a, Ix b) => (a,a) -> (a -> b)
            -> Array b c -> Array a c

instance Functor (Array a) where...
instance (Ix a, Eq b) => Eq (Array a b) where...
instance (Ix a, Ord b) => Ord (Array a b) where...
instance (Ix a, Show a, Show b)
          => Show (Array a b) where...
instance (Ix a, Read a, Read b)
          => Read (Array a b) where...
Arrays: An Abstract Data Type (Cont’d)

See also:

Chapter 8: Further Reading (1)


Chapter 8: Further Reading (2)


Chapter 8: Further Reading (3)


Chapter 8: Further Reading (4)


Chapter 9

Monoids
The Type Class Monoid

Monoids are instances of the type class Monoid:

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

...where the implementations of the monoid operations need to satisfy the so-called monoid laws.

Intuitively:

- A monoid is made up of an associative binary function `mappend`, and an element `mempty` that acts as an identity for with respect to the function `mappend`.
- The function `mconcat` takes a list of monoid values and reduces them to a single monoid value by using `mappend`. 
The Laws of Monoid

Members of the type class **Monoid** must satisfy the following three laws:

\[
\begin{align*}
\text{mempty 'mappend' x} &= x & \text{(MoL1)} \\
x 'mappend' \text{ mempty} &= x & \text{(MoL2)} \\
(x 'mappend' y) 'mappend' z &= x 'mappend' (y 'mappend' z) & \text{(MoL3)}
\end{align*}
\]

**Intuitively:**
- The first two laws (MoL1) and (MoL2) require that \text{mempty} is the identity with respect to \text{mappend}.
- The third law (MoL3) requires that function \text{mappend} is associative.

**Note:**
- It needs to be proven that these laws are satisfied by a concrete instance of class **Monoid**. This is a proof obligation for the programmer!
Remarks

- The element `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.
Example: The List Monoid (1)

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

Examples:

[1,2,3] ‘mappend‘ [4,5,6] −→ [1,2,3,4,5,6]
"Advanced " ‘mappend‘ "Functional " ‘mappend‘ "Programming"
  −→ "Advanced Functional Programming"
"Advanced " ‘mappend‘ ("Functional " ‘mappend‘ "Programming"
  −→ "Advanced Functional Programming")
("Advanced " ‘mappend‘ "Functional ") ‘mappend‘ "Programming"
  −→ "Advanced Functional Programming"
Example: The List Monoid (2)

More Examples:

\[ [1,2,3] \, \text{‘mappend‘ mempty} \rightarrow [1,2,3] \]

\text{mempty} :: [a] \rightarrow [\]

Note:

The monoid laws do not require commutativity of the binary operation \text{mappend}:

"Semester " \text{‘mappend‘ "Holiday"} 
\rightarrow "Semester Holiday"

but

"Holiday " \text{‘mappend‘ "Semester"} 
\rightarrow "Holiday Semester"
More Examples: Monoids on Numbers and Boolean Values

Numbers and Boolean values behave for several operations like a monoid

- *, +
- ||, &&

Hence, in the following we will use

- `newtype-declarations` for number types and Boolean values to allow several monoid instances declarations for number types and Boolean values, respectively
- `record-syntax` to obtain selector functions for free
Examples: Monoids on Numbers (1)

The Product and Sum Monoids:

```haskell
newtype Product a = Product { getProduct :: a }
  deriving (Eq, Ord, Read, Show, Bounded)

newtype Sum a = Sum { getSum :: 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)

instance Num a => Monoid (Sum a) where
  mempty    = Sum 0
  Sum x `mappend` Sum y = Sum (x+y)
```
Examples: Monoids on Numbers (2)

Examples:

getProduct $ Product 3 'mappend' Product 7 ->> 21
getSum $ Sum 17 'mappend' Sum 4 ->> 21

getProduct $ Product 3 'mappend' Product 7
  'mappend' Product 11 ->> 231
getSum $ Sum 3 'mappend' Sum 7 'mappend' Sum 11
  ->> 21

getSum . mconcat . map Sum $ [3,7,11] ->> 21

Product 3 'mappend' mempty ->> Product 3
getSum $ mempty 'mappend' Sum 3 ->> 3
Examples: Monoids on Boolean Values (1)

The Any and All Monoids:

```haskell
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)

newtype All = All { getAll :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid Any where
    mempty  = Any False
    Any x `mappend` Any y = Any (x || y)
    -- Any because True if any argument is true.

instance Monoid All where
    mempty  = All True
    All x `mappend` All y = All (x && y)
    -- All because True if all arguments are true.
```

Examples: Monoids on Boolean Values (2)
Examples: Monoids on Boolean Values (2)

Examples:

getAny $ Any True 'mappend' Any False ->> True
getAll $ All True 'mappend' All False ->> False

getAny $ mempty 'mappend' Any False ->> False
getAll $ All True 'mappend' mempty ->> True

getAny . mconcat . map Any $ [False,True,False,False] ->> True
getAll . mconcat . map All $ [False,True,True,False] ->> False
Remark

- For the Product, Sum, Any, and All monoids the binary function `mappend` happens to be commutative, too.
- For most instances of the type class `Monoid`, however, this does not (and need not) to hold. One such example is the Ordering monoid.
A Final Example: The Ordering Monoid (1)

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

Note:

- The definition of the binary function \texttt{mappend} leads to ‘alphabetically’ comparing lists of arguments.
- For the Ordering monoid the binary function \texttt{mappend} fails to be commutative:

\begin{align*}
\text{LT 'mappend' GT} & \rightarrow \text{LT} \\
\text{GT 'mappend' LT} & \rightarrow \text{GT}
\end{align*}
A Final Example: The Ordering Monoid (2)

Example:

The declaration

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y
    = let a = length x `compare` length y
       -- fst priority
       b = x `compare` y  -- snd priority
       in if a == EQ then b else a
```

can equivalently be rewritten as

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

by using the monoid properties.
As expected we get

```
> lengthCompare "his" "ants"  
LT
```
(since “his” is shorter than “ants”) but

```
> lengthCompare "his" "ant"  
GT
```
(since “his” is lexicographically larger than “ant”).
A Final Example: The Ordering Monoid (4)

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) -- fst priority
    'mappend' (vowels x `compare` vowels y)
    -- snd priority
    'mappend' (x `compare` y) -- thd 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
```
Summing up (1)

Monoids are especially useful for defining

- folds over various data structures.

This seems obvious for

- lists

but also holds for many other data structures including

- trees

and many others.
Summing up (2)

This has led to the introduction of the type class `Foldable` (see module `Data.Foldable`):

```haskell
class Foldable f where
    foldr :: (a -> b -> b) -> b -> f a -> b
    foldl :: (a -> b -> a) -> a -> f b -> a
    ...
```

whose fold operations generalize those on lists to foldable types, i.e., instances of the class `Foldable`:

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

...bringing us from type classes to type constructor classes.
Chapter 9: Further Reading


Chapter 10

Functors
In Chapter 7 of LVA 185.A03 we were going

- from functions to higher-order functions

In this chapter we are going

- from type classes to higher-order type classes
Chapter 9.1

Motivation
Funktionale Abstraktion höherer Stufe (1)


Betrachte folgende Beispiele:

- **Fakultätsfunktion:**
  
  \[
  
  \text{fac } n \mid n = 0 \quad = 1 \\
  \mid n > 0 \quad = n \times \text{fac} \ (n-1)
  
  \]

- **Summe der \( n \) ersten natürlichen Zahlen:**
  
  \[
  
  \text{natSum } n \mid n = 0 \quad = 0 \\
  \mid n > 0 \quad = n + \text{natSum} \ (n-1)
  
  \]

- **Summe der \( n \) ersten natürlichen Quadratzahlen:**
  
  \[
  
  \text{natSquSum } n \mid n = 0 \quad = 0 \\
  \mid n > 0 \quad = n\times n + \text{natSquSum} \ (n-1)
  
  \]
Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (2)

Beobachtung:

- Die Definitionen von \texttt{fac}, \texttt{sumNat} und \texttt{sumSquNat} folgen demselben Rekursionsschema.

Dieses zugrundeliegende gemeinsame Rekursionsschema ist gekennzeichnet durch:

- Festlegung eines Wertes der Funktion im Basisfall
- verbleibenden rekursiven Fall als Kombination des Argumentwerts \( n \) und des Funktionswerts für \( n-1 \)
Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (3)

Dies legt nahe:

▶ Obiges Rekursionsschema, gekennzeichnet durch Basisfall und Funktion zur Kombination von Werten, herauszuziehen (zu abstrahieren) und musterhaft zu realisieren.

Wir erhalten:

▶ Realisierung des Rekursionsschemas

```
recScheme base comb n
  | n==0   = base
  | n>0    = comb n (recScheme base comb (n-1))
```

1/873

685/1368
Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (4)

Funktionale Abstraktion höherer Stufe:

fac \ n = \text{recScheme} 1 (\ast) \ n

natSum \ n = \text{recScheme} 0 (+) \ n

natSquSum \ n = \text{recScheme} 0 (\lambda x \ y \rightarrow x^2 + y) \ n

Noch einfacher: In argumentfreier Ausführung

fac = \text{recScheme} 1 (\ast)

natSum = \text{recScheme} 0 (+)

natSquSum = \text{recScheme} 0 (\lambda x \ y \rightarrow x^2 + y)
Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (5)

Unmittelbarer Vorteil obigen Vorgehens:

▶ Wiederverwendung und dadurch
  ▶ kürzerer, verlässlicherer, wartungsfreundlicher Code

Erforderlich für erfolgreiches Gelingen:

▶ Funktionen höherer Ordnung; kürzer: Funktionale.

Intuition: Funktionale sind (spezielle) Funktionen, die Funktionen als Argumente erwarten und/oder als Resultat zurückgeben.
Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (6)

Illustriert am obigen Beispiel:

- Die Untersuchung des Typs von `recScheme`

\[
\text{recScheme} :: \text{Int} \to (\text{Int} \to \text{Int} \to \text{Int}) \to \text{Int}
\]

zeigt:

- `recScheme` ist ein Funktional!

In der Anwendungssituation des Beispiels gilt weiter:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fac</code></td>
<td>1</td>
</tr>
<tr>
<td><code>natSum</code></td>
<td>0</td>
</tr>
<tr>
<td><code>natSquSum</code></td>
<td>0</td>
</tr>
</tbody>
</table>
Let’s switch to a slightly more complex example

The higher-order function \texttt{map} on

- Lists

\begin{align*}
\text{mapList} & \:: \ (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{mapList} \ g \ [a] & = [a] \\
\text{mapList} \ g \ (l:ls) & = g \ l : \text{mapList} \ g \ ls
\end{align*}

- (Binary) Trees

\begin{align*}
\text{data Tree} \ a & = \text{Leaf} \ a \mid \text{Node} \ a \ (	ext{Tree} \ a) \ (\text{Tree} \ a) \\
\text{mapTree} & \:: \ (a \rightarrow b) \rightarrow \text{Tree} \ a \rightarrow \text{Tree} \ b \\
\text{mapTree} \ g \ (\text{Leaf} \ v) & = \text{Leaf} \ (g \ v) \\
\text{mapTree} \ g \ (\text{Node} \ v \ l \ r) & = \text{Node} \ (g \ v) \ (\text{mapTree} \ g \ l) \ (\text{mapTree} \ g \ r)
\end{align*}
From Higher-Order Functions

...to Higher-Order Type Classes.

It is worth noting that the implementations of

- `mapList`
- `mapTree`

like the implementations of `fac`, `natSum`, and `natSquSum` are structurally similar, too.

This similarity suggests

- striving for a function `genericMap` that covers `mapList`, `mapTree`, and more

...and leads us to the

- (type) constructor class `Functor`. 
Chapter 10.2

Constructor Class Functor
The Constructor Class Functor

Functors are instances of the constructor class Functor:

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

...where the implementation of the functor operation `fmap` needs to satisfy the so-called functor laws.

Note:

- The argument `f` of `Functor` is applied to type variables. This means, `f` is not a type variable but a type constructor that is applied to the type variables `a` and `b`.
- Members of (type) constructor classes are type constructors, no types.
- The functor operation of an instance of `Functor` takes a polymorphic function `g :: a -> b` and yields a polymorphic function `g' :: f a -> f b`, e.g., `g :: Int -> String`, and `g' :: Month Int -> Month String`. 
The Type Class Eq

For comparison recall the type class Eq:

class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)
  x == y = not (x/=y)

Note:

- The argument a of Eq is a type variable. Functions declared in Eq operate on a; a itself operates on nothing.
- This holds as well for the other type classes we considered so far such as Ord, Num, Fractional, etc.
Constructor Classes vs. Type Classes

In principle, these are similar concepts but with different members.

- **Constructor classes** (Functor, Monad, ...)
  - have **type constructors** (e.g., Tree, [], (,), ...) as members.

- **Type classes** (Eq a, Ord a, Num a, ...)
  - have **types** (e.g., Tree a, [a], (a,a), ...) as members.

Type constructors are

- functions, which from given types construct new ones.

Examples: Tuple constructors (,), (,,), (,,); List constructor []; Functional constructor ->; Input/output constructor IO,
The Laws of Functor

Members of the constructor class \texttt{Functor} must satisfy the following two \texttt{laws}:\[
\begin{align*}
\text{fmap id} & = \text{id} \quad \text{(FL1)} \\
\text{fmap (g.h)} & = \text{fmap g} \cdot \text{fmap h} \quad \text{(FL2)}
\end{align*}
\]

Intuitively:

- The “shape of the container type” is preserved.
- The contents of the container is not regrouped.

Note:

- It needs to be proven that these two laws are satisfied by a concrete instance of class \texttt{Functor} such as trees, lists, etc. This is a \texttt{proof obligation} for the programmer!
Lists and Trees as Instances of Functor (1)

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

instance Functor Tree where
    fmap g (Leaf v) = Leaf (g v)
    fmap g (Node v l r)
        = Node (g v) (fmap g l) (fmap g r)

Note:

- The symbol [] is used above in two roles, as a
  - type constructor in the line instance Functor [] where...
  - value of some list type in the line fmap g [] = [].
- The declarations instance Functor [a] where... and
  instance Functor (Tree a) where... were incorrect, since
  [a] and (Tree a) are types, no type constructors.
The next instance declarations are equivalent but more concise:

```haskell
instance Functor [] where
  fmap = mapList  -- user-defined mapList

instance Functor [] where
  fmap = map       -- predefined map

instance Functor Tree where
  fmap = mapTree   -- user-defined mapTree
```
Lists and Trees as Instances of Functor (3)

Applications:

t = Node 2 (Node 3 (Leaf 5) (Leaf 7)) (Leaf 11)

fmap (*2) t
-\rightarrow Node 4 (Node 6 (Leaf 10) (Leaf 14)) (Leaf 22)

fmap (^3) t
-\rightarrow Node 8 (Node 27 (Leaf 125) (Leaf 343))
   (Leaf 1331)

fmap (*2) [1..5] -\rightarrow [2,4,6,8,10]

fmap (^3) [1..5] -\rightarrow [1,8,27,64,125]
The function \textit{fmap} of constructor class \texttt{Functor} is
\begin{itemize}
\item the function \texttt{genericMap}
\end{itemize}
that we were looking and striving for.

Members of the constructor class \texttt{Functor} can be both
\begin{itemize}
\item pre-defined and user-defined \textit{type constructors}.
\end{itemize}
Predefined Type Constructors

Examples of predefined type constructors:

- \(( , )\), \(( , , )\), \(( , , , )\), etc.: constructors for tuple types
- \([\ ]\): constructor for list types
- \((\rightarrow)\): constructor for functional types
Notational Remarks

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
- \(f \rightarrow g\) is equivalent to \((\rightarrow )\) f g
- \(T \ a \ b\) is equivalent to \(((T \ a) \ b)\) (i.e., associativity to the left as for function application)
Illustration (1)

The signatures of the functions \texttt{fac} and \texttt{list2pair}

\begin{verbatim}
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

list2pair :: [a] -> (a,a)
list2pair (x : (y : _ )) = (x,y)
list2pair (x : _) = (x,x)
\end{verbatim}
Illustration (2)

...can equivalently be specified as follows:

```haskell
fac :: (->) Int Int
list2pair :: [] a -> (a, a)
list2pair :: [a] -> (,) a a
list2pair :: (->) [a] (a, a)
list2pair :: [] a -> (,) a a
...
list2pair :: (->) ([] a) ((,) a a)
```

Nonetheless, more easily understandable (maybe only because we are more accustomed to) seem the “classical” variants:

```haskell
fac :: Int -> Int
list2pair :: [a] -> (a, a)
```
More Examples: Maybe as Functor

data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing  = Nothing

Example:

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

  fmap (++ "Programming") Nothing
  --> Nothing
More Examples: IO as Functor (1)

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

Example:

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

is equivalent to

main =
    do line <- getLine
       let line’ = reverse line
       putStrLn $ "You said " ++ line’ ++ " backwards!"
       putStrLn $ "Yes, you said " ++ line’ ++ " backwards!"
More Examples: IO as Functor (2)

import Data.Char
import Data.List

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

has the effect of

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

Applied to

hello there

we get

E-R-E-H-T- -O-L-L-E-H
More Examples: Either as Functor (1)

    data Either a b = Left a | Right b

*Either* has two type parameters. Hence, only *(Either a)* can be made an instance of *Functor*:

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

Example:

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

    fmap (length) (Left "Programming")
        --> Left "Programming"
More Examples: Either as Functor (2)

Note that

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

would not be meaningful. Think about why not. Think about what this would mean for the types replaced for `a` and `b`. 
An Antiexample (1)

Consider the type `CounterMaybe`

\[
\text{data CounterMaybe } a = \text{CNothing}
\quad | \quad \text{CJust Int a deriving (Show)}
\]

and make it an instance of class `Functor`:

\[
\text{instance Functor CounterMaybe where}
\quad \text{fmap } f \text{ CNothing } = \text{CNothing}
\quad \text{fmap } f \text{ (CJust counter x) } = \text{CJust (counter+1) (f x)}
\]

We will show:

- The functor instance of `CounterMaybe` does not satisfy all functor laws. In this sense it is an antiexample.
An Antiexample (2)

We get:

\[\text{CNothing} \rightarrow\rightarrow \text{CNothing}\]
\[\text{CJust 0 "haha"} \rightarrow\rightarrow \text{CJust 0 "haha"}\]
\[\text{CNothing} :: \text{CMaybe a}\]
\[\text{CJust 0 "haha"} :: \text{CMaybe [Char]}\]
\[\text{CJust 100 [1,2,3]} \rightarrow\rightarrow \text{CJust 100 [1,2,3]}\]

We also get:

\[\text{fmap (++) "ha" (CJust 0 "ho"))}\]
\[\quad \rightarrow\rightarrow \text{CJust 1 "hoha"}\]
\[\text{fmap (++) "he" (fmap (++) "ha" (CJust 0 "ho"))}\]
\[\quad \rightarrow\rightarrow \text{CJust 2 "hohahe"}\]
\[\text{fmap (++) "blah"} \text{CNothing}\]
\[\quad \rightarrow\rightarrow \text{CNothing}\]
An Antiexample (3)

However, we get

\[
\text{fmap id (CJust 0 "haha")} \\
\quad \rightarrow \rightarrow \text{CJust 1 "haha"}
\]

whereas

\[
\text{id (CJust 0 "haha")} \\
\quad \rightarrow \rightarrow \text{CJust 0 "haha"}
\]

▶ This shows that \text{fmap} defined for \text{CounterMaybe} violates the first \text{Functor} law: \text{fmap id = id}

▶ \text{CounterMaybe} can thus not be considered a valid instance of class \text{Functor}. 
Summing up (1)

The constructor class Functor:

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

Laws of Functor:

- \( \text{fmap id} = \text{id} \) \((\text{FL1})\)
- \( \text{fmap (g . h)} = \text{fmap g . fmap h} \) \((\text{FL2})\)
Summing up (2)

Some instance declarations:

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)

instance Functor [] where
  fmap g [] = []
  fmap g (l:ls) = g l : fmap g ls
More concise instance declarations:

instance Functor [] where
    fmap = mapList -- user-defined mapList

instance Functor [] where
    fmap = map -- predefined map

instance Functor Tree where
    fmap = mapTree -- user-defined mapTree
Summing up (4)

Some applications of the Functor function fmap:

\[
t = \text{Node 2 (Node 3 (Leaf 5) (Leaf 7)) (Leaf 11)}
\]

\[
fmap (\times 2) t \\
\quad \rightarrow \rightarrow \text{Node 4 (Node 6 (Leaf 10) (Leaf 14)) (Leaf 22)}
\]

\[
fmap (\times 3) t \\
\quad \rightarrow \rightarrow \text{Node 8 (Node 27 (Leaf 125) (Leaf 343)) (Leaf 1331)}
\]

\[
fmap (3^\times) t \\
\quad \rightarrow \rightarrow \text{Node 9 (Node 27 (Leaf 243) (Leaf 2187)) (Leaf 177147)}
\]

\[
fmap (\times 2) [1..5] \rightarrow [2,4,6,8,10]
\]

\[
fmap (^3) [1..5] \rightarrow [1,8,27,64,125]
\]

\[
fmap (3^\times) [1..5] \rightarrow [3,9,27,81,243]
\]
Chapter 10.3

Applicative Functors
Motivation

Comparing

data Either a b = Left a | Right b

and

(->) r l

suggests that

((->) r)

can be made an instance of class Functor just as

(Either a)

can be made.

⇝ This leads us to applicative functors, i.e., to functions as functors.
Functions as Functors (1)

instance Functor ((->) r) where
  fmap f g = (\x -> f (g x))

The type of fmap

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

for this instance of Functor becomes

fmap :: (a -> b) -> ((->) r a) -> ((->) r b)

Using infix notation for -> this becomes:

fmap :: (a -> b) -> (r -> a) -> (r -> b)
Functions as Functors (2)

In effect, this means:

\[ \text{fmap } f \ g = (\lambda x \rightarrow f (g \ x)) \]

stands for function composition!

Hence, the instance definition can more concisely be given by:

\[
\text{instance Functor } ((\rightarrow) \ r) \text{ where }
\text{fmap} = (.)
\]
Functions as Functors (3)

Examples:

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:

- Calling *fmap* as an infix operation emphasizes the similarity of *fmap* and function composition.
fmap and Currying (1)

Reconsidering

\[ \text{fmap} :: (\text{Functor } f) \Rightarrow (a \to b) \to f\ a \to f\ b \]

we get:

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

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

where

```
replicate :: Int -> a -> [a]
replicate n x
  | n <= 0       = []
  | otherwise    = x : replicate (n-1) x
```
fmap and Currying (2)

The previous two examples demonstrate the
▶ lifting of an \( a \rightarrow b \) function to an \( f \ a \rightarrow f \ b \) function.

This shows that \( \text{fmap} \) can be thought of in two ways:

▶ “Curried:” As a function that takes a function and a function value and then maps that function over the functor value.

▶ “Uncurried:” As a function that takes a function and lifts that function so it operates on functor values.
fmap and Currying (3)

Examples:

\[
fmap (\text{replicate 3}) \ [1,2,3,4] \\
\rightarrow [\ [1,1,1], [2,2,2], [3,3,3], [4,4,4]]
\]

\[
fmap (\text{replicate 3}) \ (\text{Just 4}) \\
\rightarrow \ \text{Just } \ [4,4,4]
\]

\[
fmap (\text{replicate 3}) \ (\text{Right } "\text{blah}") \\
\rightarrow \ \text{Right } \ ["\text{blah}", "\text{blah}", "\text{blah}" ]
\]

\[
fmap (\text{replicate 3}) \ \text{Nothing} \\
\rightarrow \ \text{Nothing}
\]

\[
fmap (\text{replicate 3}) \ (\text{Left } "\text{foo}" ) \\
\rightarrow \ \text{Left } "\text{foo}" 
\]
Towards Using Applicative Functors

From “one” (e.g. replicate 3, (*2)) to “many”-argument mapping functions...

Some Examples:

\[
\text{fmap \((*)\) (Just 3) \rightarrow> \text{Just \(((*\) 3)}\]
\]

\[
\text{fmap \((++)\) (Just "hey") :: Maybe ([Char] \rightarrow [Char])}
\]
\[
\text{fmap compare (Just ‘a‘) :: Maybe (Char \rightarrow \text{Ordering})}
\]
\[
\text{fmap compare "A LIST OF CHARS" :: [Char \rightarrow \text{Ordering}]}
\]
\[
\text{fmap \((x \ y \ z \rightarrow x + y \ / \ z)\) [3,4,5,6]}
\]
\[
\quad :: \text{(Fractional a) \rightarrow [a \rightarrow a \rightarrow a]}
\]
\[
\text{let a = fmap \((*)\) [1,2,3,4]}
\]
\[
\text{a :: [Integer \rightarrow \text{Integer}]}
\]
\[
\text{fmap \((f \rightarrow f \, 9)\) a \rightarrow> [9,18,27,36]}
\]
The Type Constructor Class Applicative

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

Intuitively

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

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

Note:

▶ f plays the role of the applicative functor

Examples:

    Just (+3) <*> Just 9  -->> Just 12
    Just (+3) <*> Just 10 -->> Just 13

    Just (++ "hello") <*> Nothing -->> Nothing
    Nothing <*> Just "hello" -->> Nothing
The Applicative Style

Examples:

pure (+) <*> Just 3 <*> Just 5 ->> Just 8
pure (+) <*> Just 3 <*> Nothing ->> Nothing
pure (+) <*> Nothing <*> Just 5 ->> Nothing

The operator (***) is left-associative:

pure (+) <*> Just 3 <*> Just 5 =
   (pure (+) <*> Just 3) <*> Just 5
Defining an Infix Alias for \texttt{fmap} (1)

\[
(\langle\$\rangle) :: (\text{Functor } f) \Rightarrow (a \to b) \to f a \to f b
\]
\[
g \langle\$\rangle x = \text{fmap} \ g \ x
\]

\textbf{Note:}

\[
(\langle\$\rangle) :: (\text{Functor } f) \Rightarrow (a \to b) \to f a \to f b
\]
\[
f \langle\$\rangle x = \text{fmap} \ f \ x
\]

would be valid as well:

- \textbf{Type variables} (like the \texttt{f} in the function declaration) are independent of \textbf{parameter names} (like the \texttt{f} in the function body) and other \textbf{value names}.
Defining an Infix Alias for `fmap` (2)

Examples:

```
(++) <$> Just "Functional " <*> Just "Programming"
    ->> Just "Functional Programming"
```
Lists [] as Applicative (1)

instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]

Examples:

pure "Hallo" :: String  ->>  ['"Hallo"']
pure "Hallo" :: Maybe String  ->>  Just "Hallo"
Lists [] as Applicative (2)

More Examples:

\[(\ast 0),(+100),(^2)\] \(\triangleright\triangleright\) \[1,2,3\]
\[-\triangleright\] \[0,0,0,101,102,103,1,4,9\]
\[\left(,+\right),\left(,\ast \right)\] \(\triangleright\triangleright\) \[1,2\] \(\triangleright\triangleright\) \[3,4\]
\[-\triangleright\] \[4,5,5,6,3,4,6,8\]
\[\left(+\right),\left(+\right)\] \(\triangleright\triangleright\) \["ha","heh","hmm"\] \(\triangleright\triangleright\) \["?","!","."\]
\[-\triangleright\] \["ha?","ha!","ha.","heh?","heh!","heh.",
"hmm?","hmm!","hmm."\]

Also list comprehension can be replaced this way:

\[x\ast y \mid x \leftarrow \left[2,5,10\right], y \leftarrow \left[8,10,11\right]\]
\[-\triangleright\] \[16,20,22,40,50,55,80,100,110\]
\[\ast\] \(\triangleright\triangleright\) \[2,5,10\] \(\triangleright\triangleright\) \[8,10,11\]
\[-\triangleright\] \[16,20,22,40,50,55,80,100,110\]
filter (>50) \$ (\ast) \(\triangleright\triangleright\) \[2,5,10\] \(\triangleright\triangleright\) \[8,10,11\]
\[-\triangleright\] \[55,80,100,110\]
IO as Applicative (1)

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

More Examples:

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

myAction :: IO String
myAction = (++) <$> getLine <*> getLine
IO as Applicative (2)

main = do
  a <- (++) <$> getLine <*> getLine
  putStrLn $ "The concatenation of the two lines is: " ++ a
Functions (->) r as Applicative (1)

instance Applicative ((->) r) where
  pure x = (_ -> x)
  f <*> g = \x -> f x (g x)

Examples:

(pure 3) "Hello" ->> 3
pure 3 "Hello" ->> 3 (left-associativity)

(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
(+) <$> (+3) <*> (*100) $ 5 ->> 508

(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
  ->> [8.0,10.0,2.5]
Zip Lists as Applicative (1)

```haskell
data ZipList a = ZipList [a]  -- required since []
  -- can not be made
  -- twice an instance
  -- of a class like
  -- Applicative

instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs =
    ZipList (zipWith (\f x -> f x) fs xs)
```

Intuitively

- ` <*> ` applies the first function to the first value, the second function to the second value, and so on.
Zip Lists as Applicative (2)

Examples:

getZipList $
  (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
->> [101,102,103]

getZipList $
  (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
->> [101,102,103]

getZipList $
  max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
->> [5,3,3,4]

getZipList $
  (,,) <$> ZipList "dog" <*> ZipList "cat"
  <*> ZipList "rat"
->> [('d','c','r'),('o','a','a'),('g','t','t')]
The Laws of Applicative

Members of the constructor class Applicative must satisfy the following laws:

\[
\begin{align*}
\text{pure id} \triangleright\triangleright v &= v & \text{(AL1)} \\
\text{pure (.)} \triangleright\triangleright u \triangleright\triangleright v \triangleright\triangleright w &= u \triangleright\triangleright (v \triangleright\triangleright w) & \text{(AL2)} \\
\text{pure f} \triangleright\triangleright \text{pure x} &= \text{pure (f x)} & \text{(AL3)} \\
\text{u} \triangleright\triangleright \text{pure y} &= \text{pure ($y$) \triangleright\triangleright u} & \text{(AL4)}
\end{align*}
\]
Useful Functions for Applicative (1)

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

Examples:

    fmap (\x -> [x]) (Just 4)    --> Just [4]
    liftA2 (:) (Just 3) (Just [4]) --> Just [3,4]
    (:) <$> Just 3 <*> Just 4       --> Just [3,4]
Useful Functions for Applicative (2)

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

sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
Chapter 10.4
Kinds of Types and Type Constructors
Kinds of Types and Type Constructors

Like values

- types and
- type constructors

have types, too.

These types are called

- kinds.
Kinds of Types

In GHCi, kinds of types (and type constructors) can be computed and displayed using the command “:\k”:

ghci> :k Int
Int :: *

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

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

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

where * (read as “star” or as “type”) indicates that the type is a concrete type.
Type Constructors

Type constructors

- take types as parameters to eventually produce concrete types.

Example:

The type constructors \texttt{Maybe}, \texttt{Either}, and \texttt{Tree}

\begin{verbatim}
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)
\end{verbatim}

produce for \texttt{a} and \texttt{b} chosen to be \texttt{Int} and \texttt{String}, respectively, the concrete types

\begin{itemize}
  \item Maybe Int
  \item Either Int String
  \item Tree Int
\end{itemize}
Kinds of Type Constructors

Like concrete types

- type constructors have types, called kinds, as well.

ghci> :k Maybe
Maybe :: * -> *

ghci> :k Either
Either :: * -> * -> *

ghci> :k Tree
Tree :: * -> *

ghci> :k (->)
(->) :: * -> * -> *
Kinds of Partially Applied Type Constructors

Like functions

- also type constructors can be partially applied.

ghci> :k Either Int
Either Int :: * -> *

ghci> :k Either Int String
Either Int String :: *
Reconsidering the definition of type class `Functor`:

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

it becomes obvious that only

- type constructors of kind `* -> *`

can be members of type class `Functor`. 

Type Constructors as Functors
Chapter 10: Further Reading (1)


Chapter 10: Further Reading (2)


Chapter 11
Monads
Monads – A mundane approach for composing functions, for

- functional composition!

The monad approach succeeds in

- linking and composing functions

whose types are incompatible and thus inappropriate to allow their

- simple functional composition.
Monads: A Suisse Knife for Programming

Monadic programming works well 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.

- **Side-effects and explicit evaluation orders**
  - Canonical scenario: Input/output operations (IO Monad).

- **Exception and error handling**
  - Maybe Monad
Illustration

Consider:

\[ a-b \quad -- \text{Evaluation order of } a \text{ and } b \text{ is not} \]
\[ -- \text{fixed. This is crucial, if input/output} \]
\[ -- \text{is involved.} \]

Monads

- allow us to explicitly specify the order, in which operations are applied; this way, they bring an imperative flavour into functional programming.

\[
\text{do } a \leftarrow \text{getInt} \quad -- \text{Evaluation order is}
\]
\[
b \leftarrow \text{getInt} \quad -- \text{explicitly fixed:}
\]
\[
\text{return } (a-b) \quad -- \text{first } a, \text{ then } b.
\]
Chapter 11.1
Motivation
Setting the Stage

Consider:

\[ f :: a \rightarrow b \]
\[ g :: b \rightarrow c \]

Functional composition for \( f \) and \( g \) works perfectly:

\[ (g \ . \ f) = g (f \ v) \]

where

\[ (g \ . \ f) :: a \rightarrow c \]
Case Study “Debugging” (1)

Objective:

- Empowering \( f \) and \( g \) such that debug-information in terms of a string is collected and output during computation.

To this end, replace \( f \) and \( g \) by two new functions \( f' \) and \( g' \):

\[
\text{type DebugInfo = String}
\]

\[
f' :: a \rightarrow (b,\text{DebugInfo})
g' :: b \rightarrow (c,\text{DebugInfo})
\]

Unfortunately:

- \( f' \) and \( g' \) cannot be composed easily: Simple functional composition does not work any longer because of incompatible argument and result types of \( f' \) and \( g' \).
Case Study “Debugging” (2)

The below \textit{ad hoc} composition works:

\begin{verbatim}
let (fResult,fInfo) = f' v 
(gResult,gIinfo) = g' fResult 
in (gResult,gInfo++fInfo)
\end{verbatim}

...but were impractical in practice as it continuously required implementing new specific composition operations.
Case Study “Debugging” (3)

Towards a more systematic approach:

- Define a new “link” function.

\[
\text{link} :: (a,\text{DebugInfo}) \rightarrow (a \rightarrow (b,\text{DebugInfo}))
\]
\[
\rightarrow (b \rightarrow \text{DebugInfo})
\]

\[
\text{link} \ (v,s) \ g = \ \text{let} \ 
\quad (\text{gResult},\text{gInfo}) = g \ v \ \text{in} \ (\text{gResult},s++)\text{gInfo}
\]

The function `link` allows us to compose \( f' \) and \( g' \) comfortably again:

\[
h' \ v = f' \ v \ '\text{link}' \ g'
\]
Making it Practical: link, unit, lift

Introduce a new identity function that is a unit for link, and a new lift function that makes each function working with link:

\[
\begin{align*}
\text{unit } v &= (v, "") \\
\text{lift } f &= \text{unit} \ . \ f
\end{align*}
\]

The functions link, unit, and lift can now be applied in concert.

Example:

\[
\begin{align*}
f \ v &= (v, "f called. ") \\
g \ v &= (v, "g called. ") \\
h \ v &= f \ v \ '\text{link}' \ g \ '\text{link}' \ (\lambda x \rightarrow (x, "done. "))
\end{align*}
\]

We obtain:

\[
\begin{align*}
h \ 5 \rightarrow&> (5, "f called. g called. done."
\end{align*}
\]

Note that functions are applied “left to right” as desired.
Case Study “Random Numbers” (1)

The library `Data.Random` provides a function

\[
\text{random} :: \text{StdGen} \rightarrow (a,\text{StdGen})
\]

for computing (pseudo) random numbers.

Ordinary functions can use random numbers, if they can (additionally) manage a value of type `StdGen` that can be used by the next operation to generate a random number:

\[
f :: a \rightarrow \text{StdGen} \rightarrow (b,\text{StdGen})
\]

Problem:

- How to compose functions \( f \) and \( g \)?

\[
f :: a \rightarrow \text{StdGen} \rightarrow (b,\text{StdGen})
\]

\[
g :: b \rightarrow \text{StdGen} \rightarrow (c,\text{StdGen})
\]
Case Study “Random Numbers” (2)

An *ad hoc* composition:

\[
\begin{align*}
h &: a \rightarrow \text{StdGen} \rightarrow (c,\text{StdGen}) \\
h v \text{ gen} &= \text{let} \\
&(f\text{Result},f\text{Gen}) = f v \text{ gen} \text{ in } g f\text{Result} f\text{Gen}
\end{align*}
\]

More appropriate:

- The trio of functions *link*, *unit*, *lift*.

\[
\begin{align*}
\text{link} &: (\text{StdGen} \rightarrow (a,\text{StdGen})) \rightarrow \\
&(a \rightarrow \text{StdGen} \rightarrow (b,\text{StdGen})) \rightarrow \\
&\text{StdGen} \rightarrow (b,\text{StdGen}) \\
\text{link} &: g f \text{ gen} = \text{let} (v,\text{gen'}) = g \text{ gen} \text{ in } f v \text{ gen'}
\end{align*}
\]

\[
\begin{align*}
\text{unit} v \text{ gen} &= (v,\text{gen}) \\
\text{lift} f &= \text{unit} . f
\end{align*}
\]
Quintessence

The previous examples enjoy

- a common structure.

This common structure can be encapsulated in a

- new (type) constructor class.

This type class will be the (constructor) class

- Monad.
Prospect: The Constructor Class Monad

```haskell
data Debug a = D (a, String)
data Random a = R (StdGen -> (a, StdGen))

class Monad m where

  -- link
  (>>=) :: m a -> (a -> m b) -> m b

  -- link but ignore the result component of the first function
  (>>) :: m a -> m b -> m b

  -- neutral element wrt (>>=)
  return :: a -> m a

  fail :: String -> m a

  -- default implementation
  m >>= k = f >>= _ -> k
  fail = error
```
Prospect: Instance Declaration for Random

The instance declaration for type constructor Random:

```haskell
instance Monad Random where
  (R m) >>= f = R $ \gen -> (let
    (a,gen') = m gen
    (R b) = f a in b gen')

  return x = R $ \gen -> (x,gen)
```
Chapter 11.2
Constructor Class Monad
The Constructor Class Monad

Monads are instances of the 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

    m >>= k = m >>= _ -> k -- default implementation
    fail s = error s -- default implementation:
        -- represents a failing
        -- computation that outputs
        -- the error message s

...where the implementations of the monad operations (>>=), (>>), return, fail must satisfy the so-called monad laws.
```
The Laws of Monad

Members of the constructor class Monad must satisfy the following three laws:

\[
\begin{align*}
\text{return } a & \gg= f = f a \quad \text{(ML1)} \\
\text{c } \gg= \text{return} & = c \quad \text{(ML2)} \\
\text{c } \gg= (\lambda x \rightarrow (f x) \gg= g) & = (c \gg= f) \gg= g \quad \text{(ML3)}
\end{align*}
\]

Intuitively:

- return passes the value without any other effect; return is unit of (\gg=).
- sequencings given by (\gg=) do not depend on how they are bracketed; (\gg=) is associative.

Note:

- It needs to be proven that these laws are satisfied by a concrete instance of class Monad such as trees, lists, etc. This is a proof obligation for the programmer!
The Laws of Monad in Terms of (>@@>) (1)

The derived operation (>@@>) makes the intuitive meaning of the monad laws more obvious; i.e. as obvious as associativity is for the (>>>) operation:

\[ c_1 \gg (c_2 \gg c_3) = (c_1 \gg c_2) \gg c_3 \]

(Note: Associativity of (>>>) is implied by that of (>>=).)

The operation (>@@>) is defined by:

\[
(\gg) :: \text{Monad } m \Rightarrow (a \to m b) \to (b \to m c) \to (a \to m c)
\]

\[ f \gg g = \lambda x \to (f \ x) \gg= g \]
The Laws of Monad in Terms of (>@>) (2)

The monad laws in terms of (>@>):

\[
\begin{align*}
\text{return} & \ >@> \ f = f & \text{(ML1')} \\
\text{f} & \ >@> \ \text{return} = f & \text{(ML2')} \\
\text{f} \ >@> \ (\text{g} \ >@> \ h) & = \text{f} \ >@> \ (\text{g} \ >@> \ h) & \text{(ML3')} \\
\end{align*}
\]

Intuitively

- (ML1'), (ML2'): return is unit of (>@>).
- (ML3'): (>@>) is associative.

Note: As mentioned before, the above properties need to be ensured by the instance declaration. They do not hold per se.
Syntactic Sugar: The do-Notation

Monadic operations

- allow to specify the sequencing of operations explicitly.

This introduces

- an imperative flavour into functional programming.

The syntactic sugar of the so-called

- do-notation

makes this flavour more explicit.
do-Notation: A Useful Notational Variant (1)

The do-notation makes composing monadic operations syntactically more concise.

Four transformation rules

1. allow to convert compositions of monadic operations into equivalent (\(\leftrightarrow\)) do-blocks and vice versa.

(R1) \(\text{do } e \leftrightarrow e\)

(R2) \(\text{do } e_1; e_2; \ldots; e_n \leftrightarrow e_1 \gg= \_ \rightarrow \text{do } e_2; \ldots; e_n\)

(R3) \(\text{do let decllist}; e_2; \ldots; e_n \leftrightarrow \text{let decllist in do } e_2; \ldots; e_n\)

(R4) \(\text{do pattern } \leftarrow e_1; e_2; \ldots; e_n \leftrightarrow\)

\(\text{let ok pattern } = \text{do } e_2; \ldots; e_n\)

\(\text{ok } _\_ = \text{fail } "\ldots"\)

\(\text{in } e_1 \gg= \text{ok}\)
do-Notation: A Useful Notational Variant (2)

A special case of the “pattern rule” (R4):

\[(R4') \text{ do } x \leftarrow e_1; e_2; \ldots; e_n \leftrightarrow \]
\[e_1 \gg= \lambda x \rightarrow \text{do } e_2; \ldots; e_n\]

Remarks:

- (R2): If the return value of an operation is not needed, it can be moved to the front.
- (R3): A \texttt{let}-expression storing a value can be placed in front of the \texttt{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 \texttt{fail}, if the pattern matching fails.

Note: It is rule (R4) that necessitates \texttt{fail} as a monadic operation in \texttt{Monad}. Overwriting this operation allows a monad-specific exception and error handling.
Illustrating the do-Notation

...using the monad laws as example.

- The monad laws using the monadic operations:

  \[
  \text{return } a \gg= f = f \ a \quad (\text{ML1}) \\
  c \gg= \text{return} = c \quad (\text{ML2}) \\
  c \gg= (\lambda x \to (f \ x) \gg= g) = (c \gg= f) \gg= g \quad (\text{ML3})
  \]

- The monad laws using the do-notation:

  \[
  \text{do } x \leftarrow \text{return } a; \ f \ x = f \ a \quad (\text{ML1}) \\
  \text{do } x \leftarrow c; \ \text{return } x = c \quad (\text{ML2}) \\
  \text{do } x \leftarrow c; \ y \leftarrow f \ x; \ g \ y = \\
  \quad \text{do } y \leftarrow \text{(do } x \leftarrow c; \ f \ x) ; \ g \ y \quad (\text{ML3})
  \]
Quintessence: The 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

m >> k = m >>= \_ -> k -- default implementation
fail s = error s -- default implementation

Intuitively:

Monad operations

- describe actions with side effects.
- allow to fix the order of evaluation steps.
- support an imperative-like programming style w/out breaking the functional paradigm.
Quintessence: Monadic Operations

Intuitively

- **(>>=)**: The sequence operator (read as *then* (following Simon Thompson) or *bind* (following Paul Hudak)), or – maybe – as *link*.

- **return**: Returns a value w/out any other effect.

- **(>>)**: From (>>=) derived sequence operator (read as *sequence* (according to Paul Hudak)).

- **fail**: Exception and error handling.
Useful Supporting Functions for Monads

sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
    where mcons p q = do l <- p
                   ls <- q
                   return (l:ls)

sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>>) (return ())

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)

(=<<) :: Monad m => (a -> m b) -> m a -> m b
f =<< x = x >>= f
A Law linking Classes Monad and Functor

Type constructors that are an instance of both
  ▶ class Monad and class Functor
must satisfy the law:

\[
\text{fmap } g \text{ xs} = \text{xs } >>= \text{ return } . \ g \\
( = \text{ do } x \leftarrow \text{xs}; \text{ return } (g x) )
\]
Chapter 11.3
Predefined Monads
Predefined Monads

A selection of predefined monads in Haskell:

- Identity monad
- List monad
- Maybe monad
- State monad
The Identity Monad (1)

The **identity monad**, conceptually the simplest monad:

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

instance Monad Id where
  (Id x) >>= f = f x
  return = Id
```

**Note:**

- `(>>=)` and `fail` are implicitly defined by their default implementations.
The Identity Monad (2)

Remarks:

▶ 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.
▶ The operation \( (>@>) \) becomes for the identity monad forward composition of functions, i.e., \( (> >) \):

\[
(> >) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)
\]

\[
g >.> f = f . g
\]

▶ Forward composition of functions \( (> >) \) is associative with unit \( \text{id} \).
The list monad:

```
instance Monad [] where
    xs >>= f = concat (map f xs)
    return x = [x]
    fail s   = []
```

where `concat` is from the Standard Prelude:

```
concat :: [[a]] -> [a]
concat lss = foldr (++) [] lss
```
The List Monad (2)

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 monadic operations ( >>= ) and return have the types:
  - `(>>=) :: [a] -> (a -> [b]) -> [b]`
  - `return :: a -> [a]`
The List Monad (3)

The list monad is closely related to list comprehension:

\[
\text{do } x \leftarrow [1,2,3] \\
y \leftarrow [4,5,6] \\
\text{return } (x,y) \\
\Rightarrow [(1,4),(1,5),(1,6),(2,4),(2,5), \\
(2,6),(3,4),(3,5),(3,6)]
\]

Hence, the following notations are equivalent:

\[
[(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5,6]] \iff \\
\text{do } x \leftarrow [1,2,3] \\
y \leftarrow [4,5,6] \\
\text{return } (x,y)
\]

List comprehension is syntactic sugar for monadic syntax!
**The List Monad (4)**

List comprehension: Syntactic sugar for monadic syntax.

We have:

\[
[f \ x \mid x <- xs] \leftrightarrow \text{do } x <- xs; \ return \ (f \ x)
\]

\[
[a \mid a <- as, p \ a] \leftrightarrow \\
\quad \text{do } a <- as; \ if \ (p \ a) \ then \ return \ a \ else \ fail ""
\]
The Maybe Monad (1)

The **Maybe monad**:

```haskell
data Maybe a = Nothing | Just a

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= k = Nothing
  return = Just
  fail s = Nothing
```

**Remark:**

- The **Maybe** monad is useful for computation (sequences) that might produce a result, but might also produce an error.
The Maybe Monad (2)

For the Maybe monad the monadic operations (>>=) and return have the types:

\[
(\gg\gg=) \:: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b
\]
\[
\text{return} \:: a \rightarrow \text{Maybe } a
\]

The Maybe type is also a predefined member of the Functor class:

\[
\text{instance Functor Maybe where}
\]
\[
\text{fmap } f \text{ Nothing } = \text{Nothing}
\]
\[
\text{fmap } f \text{ (Just } x) = \text{Just } (f \ x)
\]
The Maybe Monad (3)

Composing functions like

\[
\begin{align*}
f & : \text{Int} \rightarrow \text{Int} \\
g & : \text{Int} \rightarrow \text{Int} \\
x & : \text{Int}
\end{align*}
\]

in \( g \, (f \, x) \) while assuming that the evaluation of \( f \) and \( g \) may fail, is possible by embedding the computation into the \text{Maybe} type:

\[
\begin{align*}
\text{case} \ (f \, x) \ & \ 
\text{of} \\
\text{Nothing} & \rightarrow \text{Nothing} \\
\text{Just} \ y & \rightarrow \text{case} \ (g \, y) \ & \ 
\text{of} \\
\text{Nothing} & \rightarrow \text{Nothing} \\
\text{Just} \ z & \rightarrow \ z
\end{align*}
\]

Though possible, this is “inconvenient.”
Embedding gets a lot easier by exploiting the membership of the Maybe type in the Maybe monad:

\[
f \ x \gg= \ y \rightarrow \ g \ y \gg= \ z \rightarrow \ return \ z
\]

which is equivalent to:

\[
do \ y <- f \ x \\
z <- g \ y \\
return \ z
\]

...the “nasty” error check is “hidden” in the Maybe monad.
The Maybe Monad (5)

Note that

\[ f \ x \ >>= \ \lambda y \to \ g \ y \ >>= \ \lambda z \to \ return \ z \]

can also be simplified to:

\[ f \ x \ >>= \ \lambda y \to \ g \ y \ >>= \ \lambda z \to \ return \ z \]

(Simplification by currying) \(\leftarrow\rightarrow\)

\[ f \ x \ >>= \ \lambda y \to \ g \ y \]

(Monad law for return) \(\leftarrow\rightarrow\)

\[ f \ x \ >>= \ g \]

(Simplification by currying) \(\leftarrow\rightarrow\)

This way, \(g \ (f \ x)\) gets \(f \ x \ >>= \ g\).
The Maybe Monad (6)

Another possibility to better cope with \((g \circ f)\) were to introduce the function:

\[
\text{composeM} :: \text{Monad } m \Rightarrow (b \rightarrow m c) \rightarrow (a \rightarrow m b) \rightarrow (a \rightarrow m c)
\]
\[
(g \ '\text{composeM}'\ f)\ x = f\ x >>= g
\]

Using \text{composeM} we obtain:

\((g \circ f)\) gets \((g \ '\text{composeM}'\ f)\)

Note:

- Both this and the previous handling of embedding the function composition of \(g\) and \(f\) into the \text{Maybe} type preserve the original notation of composing \(g\) and \(f\) in an almost 1-to-1 kind.
The State Monad (1)

Objective:
- Modelling of programs with **global (internal) state and side effects** by means of
  - functions that applied to an **initial state** yield a **final state** as part of the overall result of the computation.

The (resp. a) state monad:

```haskell
newtype State s a = St (s -> (s,a))

instance Monad (State s) where
  return x = St (\s -> (s,x))  -- The identity
  (St m) >>= f =  -- on states!
      St (\s -> let (s1,x) = m s
                St f' = f x
                in f' s1)
      -- m applied to
      -- s yields s1
      -- and x to which
      -- then f is
      -- applied to.
```

The State Monad (1)
The State Monad (2)

Intuitively

State transformers

- model and transform global (internal) states.
- are (in this setting) mappings of the type $s \rightarrow (s, a)$.
- map an initial state to a pair consisting of a (possibly modified) final state and another result component of type $a$. 
The State Monad (3)

A variant of the state monad for \( S \) a suitable fixed state type:

```haskell
data SM a = SM (S -> (S,a))

instance Monad SM where
    return a
        = SM (\s -> (s,a))
    SM sm0 >>= fsm1
        = SM $ \s0 ->
            let (s1,a1) = sm0 s0
                SM sm1 = fsm1 a1
                (s2,a2) = sm1 s1
            in (s2,a2)
```

793/1368
Predefined Monads

There are many more predefined monads in Haskell:

- Writer monad
- Reader monad
- Failure monad
- ...
- Input/output monad
The Input/Output Monad (1)

The **IO** monad:

```haskell
instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  return :: a -> IO a
```

**Intuitively:**

- `(>>=)`: If `p` and `q` are commands, then `p >>= q` is the command that first executes `p`, yielding thereby the return value `x` of type `a`, and then executes `q x`, thereby yielding the return value `y` of type `b`.

- `return`: Generates a return value w/out any input/output action.
The Input/Output Monad (2)

It is worth noting:

- The IO monad is similar in spirit to the state monad: It passes around the “state of the world.”

In more detail:

For a given suitable type \texttt{World}

- whose values represent the current state of the world

the notion of an interactive program, i.e., an IO-program, can be represented by a function of type

- \texttt{World -> World}

which may be abbreviated as:

\texttt{type IO = World -> World}
The Input/Output Monad (3)

In general:

- Interactive programs do not only modify the state of the world but may also return a result value, e.g., echoing a character that has been read from a keyboard.

This suggests to change the type of interactive programs to

```
type IO = World -> (a, World)
```
Chapter 11.4

Constructor Class MonadPlus
The Monad `MonadPlus`

...for members of `Monad` with `Null` and `Plus` operation:

```haskell
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```
The Laws of MonadPlus

Members of the constructor class **MonadPlus** must satisfy in addition to the monad laws laws for the **Null** and **Plus** operations:

**Two laws for the Null operation:**

\[
\begin{align*}
\text{m } & \gg= (x \mapsto \text{mzero}) = \text{mzero} & \text{(MPL1)} \\
\text{mzero } & \gg= \text{m} & \text{(MPL2)}
\end{align*}
\]

**Two laws for the Plus operation:**

\[
\begin{align*}
\text{m } & \text{mplus} \text{mzero} = \text{m} & \text{(MPL3)} \\
\text{mzero } & \text{mplus} \text{m} = \text{m} & \text{(MPL4)}
\end{align*}
\]

**Note:**

- As for **Functor** and **Monad**, proving the validity of the above laws for an instance of class **MonadPlus** is a proof obligation for the programmer.
Instances of MonadPlus

Instance declarations for the `Maybe` and `[]` types for the class `MonadPlus`:

```haskell
instance MonadPlus Maybe where
    mzero = Nothing
    Nothing 'mplus' ys = ys
    xs 'mplus' ys = xs

instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

**Note:**
- List concatenation `(++)` is a special case of the `mplus` operation.
- `IO` is not an instance of `MonadPlus` because of the missing null element.
Chapter 11.5
Monadic Programming
We will consider three case studies for illustration:

- **Case study I**: Summing labels of a tree.
- **Case study II**: Replacing the leaf labels of a tree by leaf labels of another type.
- **Case study III**: Replacing the labels of a tree by the number of occurrences of this label in the tree.
Case Study I

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`.

Means:

Opposing two different functional approaches:

- A classical functional approach w/out monads
- A functional approach w/ monads.
Illustration
A Functional Approach w/out Monads

1st Approach: No monads

`sTree :: Tree Int -> Int
sTree Nil       = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2`

Note:
- The order of the evaluation is **not fixed** (degrees of freedom!)
A Functional Approach w/ Monads

2nd Approach: Using the identity monad \textit{Id}

\begin{Verbatim}
sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2) = do num <- return n
    s1 <- sumTree t1
    s2 <- sumTree t2
    return (num + s1 + s2)
\end{Verbatim}

Note:

\begin{itemize}
  \item The order of the evaluation is \textit{explicitly fixed} (no degrees of freedom!)
\end{itemize}
The Identity Monad

Recall the **identity monad**:

```haskell
data Id a = Id a

instance Monad Id where
    (>>=) (Id x) f = f x
    return       = Id
```
Opposing the Two Approaches

Comparing the two approaches \textit{w/} and \textit{w/out} monads, we observe:

- Unlike \texttt{sTree}, function \texttt{sumTree} has an “imperative” flavour very similar to the sequential sequence of (imperative) assignments:

\begin{center}
\begin{tabular}{ll}
\textbf{Imperative} & \textbf{Monadic} \\
num := n; & do num <- return n \\
s1 := sumTree t1; & s1 <- sumTree t1 \\
s2 := sumTree t2; & s2 <- sumTree t2 \\
return (num + s1 + s2); & return (num + s1 + s2)
\end{tabular}
\end{center}
Another Functional Approach w/ Monads

3rd Approach: Using monad Id and an extraction function

\[
\text{extract} :: \text{Id} \ a \rightarrow a
\]
\[
\text{extract} \ (\text{Id} \ x) = x
\]

Using \text{extract} we get a function of type Tree Int \rightarrow Int:

\[
\text{extract} \ . \ \text{sumTree} :: \text{Tree} \ \text{Int} \rightarrow \text{Int}
\]

Example:

\[
(\text{extract} \ . \ \text{sumTree})
\]
\[
(\text{Node} \ 5 \ (\text{Node} \ 3 \ \text{Nil} \ \text{Nil}) \ (\text{Node} \ 7 \ \text{Nil} \ \text{Nil}))
\]
\[
\rightarrow
\]
\[
\text{extract} \ (\text{sumTree})
\]
\[
(\text{Node} \ 5 \ (\text{Node} \ 3 \ \text{Nil} \ \text{Nil}) \ (\text{Node} \ 7 \ \text{Nil} \ \text{Nil}))
\]
\[
\rightarrow
\]
\[
\text{extract} \ (\text{Id} \ 15) \rightarrow> 15
\]
Case Study II

Given:

\[
data \text{ Tree } a = \text{ Leaf } a \mid \text{ Branch (Tree } a) \ (\text{Tree } a)
\]

Objective:

- Replace the labels of the leafs that are supposed to be of type `Char` by continuous natural numbers.
Illustration

Let \texttt{test} be defined by

\[
\texttt{test} = \texttt{let } \ t = \texttt{Branch (Leaf 'a') (Leaf 'b')} \\texttt{in label (Branch } \ t \ \texttt{t)}
\]

Then \texttt{test} shall be transformed to:

\[
\text{Branch (Branch (Leaf 0) (Leaf 1))} \\
(Branch (Leaf 2) (Leaf 3))
\]
A Functional Approach w/out Monads

1st Approach: No monads

```haskell
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:

- Simple but passing the value \( n \) through the incarnations of \( \text{lab} \) is “intricate.”
A Functional Approach w/ Monads (1)

2nd Approach: Using the state monad

newtype Label a = Label (Int -> (Int,a))

...“matches” the pattern of the state monad \( SM \).

We define:

\[
\text{instance Monad Label where}
\]

\[
\qquad \text{return } a = \text{Label } (s \rightarrow (s,a))
\]
\[
\text{Label } lt0 \gg= flt1 = \text{Label } \$ \ s0 \rightarrow
\]
\[
\qquad \text{let } (s1,a1) = lt0 s0
\]
\[
\qquad \text{Label } lt1 = flt1 a1
\]
\[
\qquad \text{in } lt1 s1
\]

Note: The \( \$ \)-operator in the definition of \( (\gg=) \) can be dropped, if the expression \( s0 \rightarrow \text{let } \ldots \text{ in } lt1 s1 \) is bracketed.
A Functional Approach w/ Monads (2)

This allows solving the renaming of labels as follows:

\[
\text{mlabel} :: \text{Tree a} \rightarrow \text{Tree Int} \\
\text{mlabel } t = \text{let Label } lt = \text{mlab } t \\
\text{ in snd (lt 0)} \\
\]

\[
\text{mlab} :: \text{Tree a} \rightarrow \text{Label (Tree Int)} \\
\text{mlab } (\text{Leaf } a) \\
\quad = \text{do } n \leftarrow \text{getLabel} \\
\quad \quad \text{return } (\text{Leaf } n) \\
\text{mlab } (\text{Branch } t1 \ t2) \\
\quad = \text{do } t1' \leftarrow \text{mlab } t1 \leftarrow \text{mlab } t1 \\
\quad \quad t2' \leftarrow \text{mlab } t2 \leftarrow \text{mlab } t2 \\
\quad \quad \text{return } (\text{Branch } t1' \ t2') \\
\]

\[
\text{getLabel} :: \text{Label Int} \\
\text{getLabel} = \text{Label } (\backslash n \rightarrow (n+1,n))
\]
A Functional Approach w/ Monads (3)

Let \texttt{mtest} be defined by

\[
\texttt{mtest} = \texttt{let } t = \texttt{Branch (Leaf 'a') (Leaf 'b')} \\
\texttt{in mlabel (Branch t t)}
\]

Then we get:

- \texttt{mlabel} applied to
  \[
  \texttt{Branch (Leaf 'a') (Leaf 'b')}
  \]
  yields as desired:
  \[
  \texttt{Branch (Branch (Leaf 0) (Leaf 1))} \\
  \texttt{(Branch (Leaf 2) (Leaf 3))}
  \]
Case Study III

Given:

```haskell
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Objective:

- Replace labels of equal value that are supposed to be of type `String` by the same natural number.
A Functional Approach w/ Monads (1)

**Ultimate Goal:** A function `numTree` of type

\[
\text{numTree} :: \text{Eq a} \Rightarrow \text{Tree a} \rightarrow \text{Tree Int}
\]

solving this task with monadic programming using the state monad.

In order to eventually arrive at this function we start with:

\[
\text{numberTree} :: \text{Eq a} \Rightarrow \text{Tree a} \rightarrow \text{State a (Tree Int)}
\]

\[
\text{numberTree Nil} = \text{return Nil}
\]

\[
\text{numberTree (Node x t1 t2)} =
\]

\[
\quad \text{do num <- numberNode x}
\]

\[
\quad \text{nt1 <- numberTree t1}
\]

\[
\quad \text{nt2 <- numberTree t2}
\]

\[
\quad \text{return (Node num nt1 nt2)}
\]
A Functional Approach w/ Monads (2)

Next, we are storing pairs of the form

\((\text{<string>}, \text{<number of occurrences>})\)

in a table of type:

\[
\text{type Table a} = [a]
\]

In particular:

The table

\([\text{True, False}]\)

encodes that the value \text{True} is associated with 0 and \text{False} with 1.
A Functional Approach w/ Monads (3)

Defining the state monad we consider:

data State a b = State (Table a -> (Table a, b))

instance Monad (State a) where
(State st) >>= f
    = State (\tab -> let
                    (newTab,y) = st tab
                    (State trans) = f y
                    in
                    trans newTab)

return x = State (\tab -> (tab,x))

Intuitively:

- Values of type \textit{b}: Result of the monadic operation.
- Update of the table: Side effect of the monadic operation.
A Functional Approach w/ Monads (4)

Defining the missing function `numberNode`:

```haskell
numberNode :: Eq a => a -> State a Int
numberNode x = State (nNode x)
```

```haskell
nNode :: Eq a => a -> (Table a -> (Table a, Int))
nNode x table
  | elem x table = (table, lookup x table)
  | otherwise = (table++[x], length table)
```

-- `nNode` yields the position of `x` in the table:
-- via `lookup`, if stored in the table; after
-- adding `x` to the table via `length` otherwise

```haskell
lookup :: Eq a => a -> Table a -> Int
lookup ... (still to complete)
```
A Functional Approach w/ Monads (5)

Putting the pieces together, we get for

\[ \text{exampleTree} :: \text{Eq a} \Rightarrow \text{Tree a} : \]

\[ \text{numTree} \text{exampleTree} :: \text{State a (Tree Int)} \]

Using an extraction function we get now the desired implementation of the function \text{numTree} of type

\[ \text{numTree} :: \text{Eq a} \Rightarrow \text{Tree a} \rightarrow \text{Tree Int} : \]

\[ \text{extract} :: \text{State a b} \rightarrow \text{b} \]
\[ \text{extract} \ (\text{State st}) = \text{snd} \ (\text{st} \ [\ ]) \]

\[ \text{numTree} :: \text{Eq a} \Rightarrow \text{Tree a} \rightarrow \text{Tree Int} \]
\[ \text{numTree} = \text{extract} \ . \ \text{numberTree} \]
Chapter 11.6
Monadic Input/Output
Handling Input/Output so Far

The programs we considered so far, handle input/output monolithically, in a way that resembles

- batch processing.

In fact, there is no interaction between a program and a user:

- All input data must be provided at the very beginning.
- Once called there is no opportunity for the user to react on a program’s response and behaviour.

Peter Pepper. *Funktionale Programmierung.* Springer−Verlag, 2003, S.245
Handling Input/Output Henceforth

Our Objective:

Modifying the handling of input/output such that programs become and behave like

- (sequentially) composed dialogue components while preserving referential transparency as far as possible.

It is worth noting

As illustrated by the previous figure, input/output is

- a **major source for side effects** in a program: e.g., each read statement like `read` will usually yield a different value for each call, i.e. referential transparency is lost.
Monadic Input/Output in Haskell

Conceptually, a Haskell program consists of

- a computational core and
- an interaction component.

Monadic Input/Output

The monad concept of Haskell allows to

- distinguish (and conceptually separate) functions that belong to the
  - computational core (pure functions)
  - interaction component (impure functions, i.e. having side effects).

by assigning different types to them:

\[ \sim \text{Int, Real, String,... vs. IO Int, IO Real, IO String,...} \]
where the type constructor \( \text{IO} \) is an instance of Monad.

- specify the evaluation order of functions of the interaction component (i.e., of basic input/output primitives provided by Haskell) by explicitly using the features of monadic programming.
Recall Chapter 11.3

The Input/Output Monad (1)

The IO monad:

```haskell
instance Monad IO where
    (>>=) :: IO a -> (a -> IO b) -> IO b
    return :: a -> IO a
```

Intuitively:

- `(>>=)`: If `p` and `q` are commands, then `p >>= q` is the command that first executes `p`, yielding thereby the return value `x` of type `a`, and then executes `q x`, thereby yielding the return value `y` of type `b`.
- `return`: Generates a return value w/out any input/output action.
Recall Chapter 11.3

The Input/Output Monad (2)

It is worth noting:

- The IO monad is similar in spirit to the state monad: It passes around the “state of the world.”

In more detail:

For a given suitable type World

- whose values represent the current state of the world

the notion of an interactive program, i.e., an IO-program, can be represented by a function of type

- World -> World

which may be abbreviated as:

type IO = World -> World
Recall Chapter 11.3

The Input/Output Monad (3)

In general:

- Interactive programs do not only modify the state of the world but may also return a result value, e.g., echoing a character that has been read from a keyboard.

This suggests to change the type of interactive programs to

\[
\text{type IO} = \text{World} \rightarrow (a, \text{World})
\]
Typical Interaction Examples (1)

A simple question/response interaction with the user:

```haskell
ask :: String -> IO String
ask question = do
    putStrLn question
    getLine

interAct :: IO ()
interAct =
    do name <- ask "May I ask your name?"
       putStrLnLine ("Welcome " ++ name ++ "!")
```
Typical Interaction Examples (2)

Input/output from/to files:

```haskell
  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
    putStrLn "Please input a file name: "
    fname <- getLine
    contents <- readFile fname
    putStrLn contents
```

Typical Interaction Examples (3)

Note the relationship of the do-notation

```haskell
do writeFile "testFile.txt" "Hello File System!"
    putStr "Hello World!"
```

and the monadic operations:

```haskell
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```

Note also the (subtle) difference in the result types:

```haskell
Main>putStr ('a':(’b’:(’c’:[]))
Main>putChar (head ['x','y','z'])
```

```haskell
->> abc :: IO ()
->> x :: IO ()
```

but

```haskell
Main>('a':(’b’:(’c’[:])))
Main>head ['x','y','z']
```

```haskell
->> "abc" :: [Char]
->> ’x’ :: Char
```

```haskell
Main>print "abc"
Main>print ’x’
```

```haskell
->> "abc" :: IO ()
->> ’a’ :: IO ()
```
More Examples (1)

The output command sequence

```haskell
do writeFile "testFile.txt" "Hello File System!"
putStr "Hello World!"
```

...is equivalent to:

```haskell
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```
More Examples (2)

It is worth noting:

From

\[(\gg) : : \text{Monad}\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b\]

and

\[
\text{writeFile } "\text{testFile.txt}" \\
"\text{Hello File System!}" : : \text{IO ()} \\
\text{putStr } "\text{Hello World!}" : : \text{IO ()}
\]

...we conclude for our example that \(m = \text{IO, a = ()}, \) and \(b = ()\). Overall, we thus obtain:

\[(\gg) : : \text{IO ()} \rightarrow \text{IO ()} \rightarrow \text{IO ()}\]
More Examples (3)

Illustrating local declarations within do-constructs:

```haskell
reverse2lines :: IO ()
reverse2lines
  = do line1 <- getLine
      line2 <- getLine
      let rev1 = reverse line1
      let rev2 = reverse line2
      putStrLn rev2
      putStrLn rev1
```

is equivalent to:

```haskell
reverse2lines :: IO ()
reverse2lines
  = do line1 <- getLine
      line2 <- getLine
      putStrLn (reverse line2)
      putStrLn (reverse line1)
```
Summing up (1)

Overall, the monadic handling of input/output in Haskell renders possible:

The shift from

- “batch-like” input/output processing that works exclusively by pure functions of the computational core as illustrated below

---

Summing up (2)

...to an interactive, dialogue-oriented input/output processing w/out breaking the functional paradigm (keyword: referential transparency!)

Peter Pepper. *Funktionale Programmierung.* Springer−Verlag, 2003, S.253
Stream-based Input/Output (1)

Early versions of Haskell foresaw a stream-based handling of input/output:

- **Stream-based** considering programs functions on streams: 
  \[ \text{IOprog :: String -> String} \]

Input/output streams on terminals, file systems, printers,...
Stream-based Input/Output (2)

Advantages and disadvantages:

- **Stream-based** input/output handling for languages with
  - **eager** semantics:
    - there is **no real stream model** (the input must completely be provided and consumed at the beginning and must thus be finite); hence, input/output is limited to a batch- or stack-like processing.
  - **lazy** semantics:
    - Interactions are possible; thanks to **lazy evaluation** inputs/outputs are always in “proper” order.
    - **But:** the causal and temporal relationship between input and output is often “obscure”; special synchronization might be used to overcome that.
    - **Overall:** streambased input/output reaches its limit when switching to graphical user interfaces and random access to files.
ML-Style Input/Output

The **ML-style** of handling input/output is

- a *Unix-like handling* of display, keyboard, etc. as files: `std_in`, `std_out`, `open_in`, `open_out`, `close_in`, ...

**Advantages and disadvantages:**

- The handling is simple but at the cost of anomalies like those discussed in LVA 185.A03; in particular, referential transparency is lost.
Last but not least

Input/output handling in functional languages is an important research topic:

Chapter 11.7

A Fresh Look at the Haskell Class Hierarchy
A Section of the Haskell Class Hierarchy (1)

...including the constructor classes Monad, MonadPlus, and Functor:

```
Eq
  (==) (/=)

Show
  showsPrec
  show
  showList

Ord
  compare
  (,) (<=) (>=) (>)
  max min

Num
  (+) (−) (*)
  negate
  abs signum
  fromInteger

Enum
  succ pred
toEnum
fromEnum
enumFrom
enumFromThen
enumFromTo
denumFromThenTo

Functor
  fmap

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

MonadPlus
  mZero
  mPlus
```

Fethi Rabhi, Guy Lapalme. *Algorithms*. Addison–Wesley, 1999, Figure 2.4, p.46
A Section of the Haskell Class Hierarchy (2)

A Section of the Haskell Class Hierarchy (3)

Fethi Rabhi, Guy Lapalme. *Algorithms: A Functional Approach*, Addison-Wesley, 1999, Figure 2.4, p.46
### Selected Types and their Class Membership

<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 Functor Monad</td>
<td>Eq Ord</td>
</tr>
<tr>
<td>(a,b)</td>
<td>Read</td>
<td>Eq Ord Bounded</td>
</tr>
<tr>
<td>(→&gt;)</td>
<td></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></td>
</tr>
<tr>
<td>Char</td>
<td>Eq Ord Enum Read</td>
<td></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>RealFloat Read</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td></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 Monad</td>
<td></td>
</tr>
<tr>
<td>IOError</td>
<td>Eq</td>
<td>Eq Ord</td>
</tr>
<tr>
<td>Maybe</td>
<td>Functor Monad</td>
<td>Eq Ord Enum Read</td>
</tr>
<tr>
<td>Ordering</td>
<td></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
Last but not least (1)

Monads – where does the term come from?

Monads, a term that

- has already been used by Gottfried Wilhelm Leibniz as a counterpart to the term “atom.”
- has been introduced into programming language theory by Eugenio Moggi in the realm of category theory as a means for describing the semantics of programming languages:

Monads, a term that

- has become popular in the world of functional programming (but w/out the background from category theory), especially because monads (Philip Wadler, 1992)
  - allow to introduce some useful aspects of imperative programming into functional programming,
  - are well suited for integrating input/output into functional programming, as well as for many other application 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!
Chapter 11: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17, 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)


Chapter 12
Arrows
Motivation

The higher-order type (constructor) class

- generalizes the type class `Monad`.

and provides an even more general concept for

- composing functions.

that is particularly useful for

- functional reactive programming (cp. Chapter 15).
The Constructor Class Arrow

Arrows are instances of the constructor class Arrows:

```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)
```

Note:

- `pure` allows embedding of ordinary functions into the constructor class `Arrow`.
- `(>>>)` serves the composition of computations.
- `first` has as an analogue on the level of ordinary functions the function `firstfun` with `firstfun f = \( (x, y) \rightarrow (f x, y) \)"
The Laws of Arrow

Members of the constructor class \textit{Arrow} must satisfy the following nine laws:

\begin{align*}
\text{pure id} & > >> f = f & (AL1): \text{identity} \\
f & > >> \text{pure id} = f & (AL2): \text{identity} \\
(f > >> g) > >> h = f > >> (g > >> h) & (AL3): \text{associativity} \\
\text{pure}(g . f) & = \text{pure} f > >> \text{pure} g & (AL4): \text{functor composition} \\
\text{first}(\text{pure} f) & = \text{pure}(f \times \text{id}) & (AL5): \text{extension} \\
\text{first}(f > >> g) & = \text{first} f > >> \text{first} g & (AL6): \text{functor} \\
\text{first} f > >> \text{pure}(\text{id} \times g) & = \text{pure}(\text{id} \times g) > >> \text{first} f & (AL7): \text{exchange} \\
\text{first} f > >> \text{pure} \text{fst} & = \text{pure} \text{fst} > >> f & (AL8): \text{unit} \\
\text{first}(\text{first} f) > >> \text{pure} \text{assoc} & = \text{pure} \text{assoc} > >> \text{first} f & (AL9): \text{association}
\end{align*}
Creating Instances of Class Arrow

**Ordinary functions as instance of constructor class Arrow:**

```haskell
instance Arrow (->) where
    pure f = f
    f >>> g = g . f
    first f = f × id
```

**Note:**

- The function `first` could also be defined by:
  ```haskell
  first f = \( (b,d) \to (f b, d) \) ```
Useful Supporting Functions (1)

\[ (\times) :: (a \to a') \to (b \to b') \to (a, b) \to (a', b') \]
\[ (f \times g) (a, b) = (f a, g b) \]

\[ \text{assoc} :: ((a, b), c) \to (a, (b, c)) \]
\[ \text{assoc} \sim (\sim (x, y), z) = (x, (y, z)) \]

\[ \text{second} :: \text{Arrow} \ a \Rightarrow a \ b \ c \Rightarrow a \ (d, b) \ (d, c) \]
\[ \text{second} \ f = \text{pure} \ \text{swap} \gg \gg \text{first} \ f \gg \gg \text{pure} \ \text{swap} \]

\[ \text{swap} :: (a, b) \to (b, a) \]
\[ \text{swap} \sim (x, y) = (y, x) \]
...related to the constructor class Arrow:

\[(***)\]  \(\text{Arrow } a \rightarrow a \ b \ c \rightarrow a \ b' \ c' \rightarrow a \ (b.b') (c,c')\)

\(f *** g = \text{first } f >>> \text{second } g\)

\[(&&&)\]  \(\text{Arrow } a \rightarrow a \ b \ c \rightarrow a \ b \ c' \rightarrow a \ b \ (c,c')\)

\(f &&& g = \text{pure } (\lambda b \rightarrow (b,b)) >>> f *** g\)

\(\text{idA} \ ::= \text{Arrow } a \rightarrow a \ b \ b\)

\(\text{idA} = \text{pure } \text{id}\)
Background and Motivation (1)

Notions of computation:

add :: (b -> Int) -> (b -> Int) -> (b -> Int)
add f g z = f z + g z
Background and Motivation (2)

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:
Generalizing \texttt{add} to non-determinism:

\begin{verbatim}
    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 ]
\end{verbatim}
Generalizing \textit{add} to \textit{mapping transformers}:

\begin{verbatim}
  type MapTrans s i o = (s \rightarrow i) \rightarrow (s \rightarrow o)

  addMT :: MapTrans s b Int \rightarrow MapTrans s b Int \rightarrow MapTrans s b Int
  addMT f g m z = f m z + g m z
\end{verbatim}
Generalizing `add` to simple automata:

```
newtype Auto i o = A (i -> (o, Auto i o))

addAuto :: Auto b Int -> Auto b Int -> Auto b Int
addAuto (A f) (A g)
    = A (\z -> let (x,f') = f z
                (y,g') = g z
                in (x+y), addAuto f' g'))
```

All together, this

- allows modelling of synchronous circuits.
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 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 $B$.

Arrows yield a sufficiently general interface to describe these commonalities uniformly and to encapsulate them in a class.
Implementing the preceding examples as instances of the class `Arrow`:

```haskell
newtype State s i o = ST ((s,i) -> (s,0))

newtype NotDet 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))
```
State transformers:

instance Arrow (State s) where
    pure f = ST (id x f)
    ST f >>> ST g = ST (g . f)
    first (ST f) = ST (assoc . (f x id) . unassoc)

unassoc :: (a,(b,c)) -> ((a,b),c)
unassoc ~(x, ~(y,z)) = ((x,y),z)
Back to the Examples (4)

Non-determinism:

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])
Mapping transformers:

```haskell
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)

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)
```
Back to the Examples (6)

Simple automata:

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’))
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 (+))
```

It is worth noting:

- 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.
Chapter 12: Further Reading


Part V
Applications
Chapter 13

Parsing
Parsing: Lexical and syntactical analysis

- Combinator (composition operator) parsing
- Monadic parsing
Lexical and Syntactical Analysis

...in the following summarized as parsing.

Parsing

- an(other) application of functional programming often used to demonstrate its power and elegance.
- enjoys a long history. As an example of early work see e.g.:
Functional Implementation Approaches for Parsing

Two conceptually different implementation approaches:

- **Combinator parsing (higher-order functions parsing)**
  - \(\leadsto\) recursive descent parsing

- **Monadic parsing**
The presentation here

...is based on:

- Chapter 17

Parsing informally

The basic problem:

- **Read** a sequence of objects of type a.
- **Extract** from this sequence an object or a list of objects of type b.
Illustrating Example: Parsing of Expressions

Consider:

- **Expressions**

```haskell
data Exp = Lit Int | Var Name | Op Ops Exp Exp

data Ops = Add | Sub | Mul | Div | Mod

Op Mul (Op Add (Lit 2) (Lit 3)) (Lit 3)

corresponds to   \((2+3)*3\)
```

The parsing task to be solved:

- **Read** an expression of the form \( (2+3)*5 \) and yield/“extract” the corresponding expression of type `Exp`.

(Note: This can be considered the reverse of the `show` function. It is similar to the derived `read` function, but differs in the arguments it takes (expressions of the form \( (2+3)*5 \) vs. expressions of the form `Op Mul (Add (Lit 2) (Lit 3)) (Lit 5)`.)
Towards the Type of a Parser Function (1)

What shall be the type of a parsing function?

Naive specification of the type of a parser function:

\[
\text{type BSParse1 a b = } [a] \rightarrow b
\]

<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>
</tbody>
</table>
| number     | "234"  | 2 or 23 or 234 \?
| bracket    | "234"  | no result, failure? |

Open issues to be answered:

How shall the parser behave if there

\[\text{are multiple results?}\]

\[\text{is a failure?}\]
Towards the Type of a Parser Function (2)

First refinement of the type of a parser function:

```
type BSParse2 a b = [a] -> [b]
```

<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>

Open issue to be answered:

- What shall the parser do with the remaining input?
The Type of a Parser Function

The final specification of the type of a parser function:

\[
\text{type } \text{Parse } a \ b = [a] \rightarrow [(b, [a])]\]

<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>[('', &quot;xyz&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>
Remarks and Conventions

It is worth noting:

▶ The capability of delivering multiple results enables the analysis of ambiguous grammars

⇝ list of successes technique

▶ Each element in the output list represents a successful parse.

Convention:

▶ Delivery of the empty list: Signals failure of the analysis.

▶ Delivery of a non-empty list: Signals success of the analysis; each element of the list is a pair, whose first component is the identified object (token) and whose second component is the input not yet considered.
Chapter 13.1

Combinator Parsing
Basic Parsers (1)

Primitive, input-independent parsing functions:

- The always failing parsing function
  
  none :: Parse a b
  none inp = []

- The always successful parsing function
  
  succeed :: b -> Parse a b
  succeed val inp = [(val,inp)]

Remark:

- The `none parser` always fails. It does not accept anything.
- The `succeed parser` does not consume its input. In BNF-notation this corresponds to the symbol $\varepsilon$ representing the empty word.
Basic Parsers (2)

Primitive, input-dependent parsing functions:

- **Recognizing single objects (token):**

  ```haskell
  token :: Eq a => a -> Parse a a
  token t (x:xs)
  | t == x = [(t,xs)]
  | otherwise = []
  token t [] = []
  ```

- **Recognizing single objects satisfying a particular property:**

  ```haskell
  spot :: (a -> Bool) -> Parse a a
  spot p (x:xs)
  | p x = [(x,xs)]
  | otherwise = []
  spot p [] = []
  ```
Simple Applications of Basic Parsers

Application:

bracket = token '(

dig = spot isDigit

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')

Note: token can be defined using spot

token t = spot (== t)
Intuition and Motivation for Combining Parsers

...obtaining (more) complex (re-usable) parsing functions:

- Combinator Parsing

Objective

- Building a library of higher-order polymorphic functions, which are then used to construct parsers.
Combining Parsers – Alternatives

1) Composition of parsers as alternatives:

alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp

Underlying intuition:

▶ An expression, e.g., is either a literal, or a variable or an operator expression.

Example:

(bracket 'alt' dig) "234" ->> [] ++ [(2,"34")]

⇝ The alt parser combines the results of the parses given by the parsers p1 and p2.
Combining Parsers – Sequential Composition

2) Sequential composition of parsers:

\[
\text{infixr 5 >\*>}
\]
\[
(>\*>) :: \text{Parse } a \text{ b } \rightarrow \text{Parse } a \text{ c } \rightarrow \text{Parse } a \text{ (b,c)}
\]
\[
(>\*>) \ p1 \ p2 \ \text{inp} \\
= [((y,z),\text{rem2}) \mid (y,\text{rem1}) <- p1 \ \text{inp}, \\
(z,\text{rem2}) <- p2 \ \text{rem1} ]
\]

Underlying intuition:

- An operator expression starts with a bracket followed by a number.
Combining Parsers – Sequential Composition

Example:

Because of \textbf{number} "24(" \(\rightarrow\) [\(2,"4("), (24,"(")]\) we obtain:

\[
\begin{align*}
\text{(number } \gg\gg \text{ bracket) } "24(" \\
\rightarrow & \quad [((y,z),\text{rem2}) \mid (y,\text{rem1}) \leftarrow [\(2,"4("), (24,"(")], \\
& \quad (z,\text{rem2}) \leftarrow \text{bracket rem1 }] \\
\rightarrow & \quad [((2,z),\text{rem2}) \mid (z,\text{rem2}) \leftarrow \text{bracket } "4(" \] ++ \\
& \quad [((24,z),\text{rem2}) \mid (z,\text{rem2}) \leftarrow \text{bracket } "(" \] \\
\rightarrow & \quad [] ++ [((24,z),\text{rem2}) \mid (z,\text{rem2}) \leftarrow \text{bracket } "(" ]
\end{align*}
\]

Because of \textbf{bracket} "(" \(\rightarrow\) [\('(','"'\')]\) we finally get:

\[
\begin{align*}
\rightarrow & \quad [((24,z),\text{rem2}) \mid (z,\text{rem2}) \leftarrow [\('(','"'\])] \]
\rightarrow & \quad [((24,'('), "'\)])
\end{align*}
\]
Combining Parsers – Transformation

3) Transformation by parsers:

⇝ transform the item returned by the parser, e.g., build something from it.

\[
\text{build} :: \text{Parse a b } \rightarrow (\text{b } \rightarrow \text{ c}) \rightarrow \text{Parse a c}
\]

\[
\text{build } \ p \ f \ \text{inp} = [(f \ x, \text{rem}) | (x,\text{rem}) \leftarrow p \ \text{inp}]
\]

Example: Note, \text{digList} returns a list of numbers and shall be embedded such that the number represented by it is returned.

\[
(\text{digList 'build' digsToNum}) \ "21a3"
\]

\[
\rightarrow [(\text{digsToNum } x,\text{rem}) | (x,\text{rem}) \leftarrow \text{digList } "21a3"]
\]

\[
\rightarrow [(\text{digsToNum } x,\text{rem}) | (x,\text{rem}) \leftarrow
\quad [("2","1a3"),("21","a3")]]
\]

\[
\rightarrow [(\text{digsToNum } "2", "1a3"), (\text{digsToNum } "21", "a3")]
\]

\[
\rightarrow [(2,"1a3"), (21,"a3")]
\]
Universal Parser Basis

The Clou:

The

- basic parsers
- alt
- (>>)
- build

constitute a universal “parser basis,” i.e., allow to build any parser which might be desired.
Example: A Parser for a List of Objects

We suppose to be given a parser recognizing single objects:

\[
\text{list :: Parse } a \, b \rightarrow \text{Parse } a \, [b] \\
\text{list } p = (\text{succeed } []) \, \text{'alt'} \\
\quad ((p \, >*\, \text{list } p) \, \text{'build'} \, (\text{uncurry } (:)))
\]

Intuition:

- A list can be empty. 
  ⟹ this is recognized by the parser `succeed []`.

- A list can be non-empty, i.e., it consists of an object followed by a list of objects.
  ⟹ this is recognized by the combined parser `p >** list p`, where we use `build` to turn a pair `(x, xs)` into the list `(x:xs)`.
Summing up

...on combining parsers (parser combinators):

- Parsing functions in the above fashion are structurally similar to grammars in BNF-form. For each operator of the BNF-grammar there is a corresponding (higher-order) parsing function.

- These higher-order functions combine simple(r) parsing functions to (more) complex parsing functions.

Summary of the Universal Parser Basis (1)

Priority of the sequence operator

\texttt{infixr 5 \textgreater\textgreater}

Parser type

\texttt{type Parse a b = [a] \rightarrow [(b,[a])]}

Input-independent parsing functions

\texttt{none :: Parse a b}
\texttt{none inp = []}

\texttt{succeed :: b \rightarrow Parse a b}
\texttt{succeed val inp = [(val,inp)]]}
Summary of the Universal Parser Basis (2)

Recognizing single objects

token :: Eq a => a -> Parse a a
token t = spot (==t)

Recognizing single objects satisfying a particular property

spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x = [(x,xs)]
  | otherwise = []
spot p [] = []
Summary of the Universal Parser Basis (3)

Alternatives

alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp

Sequences

(>**>) :: 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 ]
Example

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
    ((p >>= list p) 'build' (uncurry (::)))
```
2nd Application of the Universal Parser Basis

Back to the initial example – a parser for expressions.

We consider expressions of the form:

```haskell
data Expr = Lit Int | Var Name | Op Ops Expr Expr

data Ops = Add | Sub | Mul | Div | Mod
```

Op Add (Lit 2) (Lit 3) corresponds to 2+3

where the following convention shall hold:

- **Literals**: 67, ∼89, etc., where ∼ is used for unary minus.
- **Names**: the lower case characters from 'a' to 'z'.
- **Applications of the binary operations** ...+,*,—,/,%,
  where % is used for mod and / for integer division.
- **Expressions are fully bracketed**; white space is not permitted.
The parser

```haskell
parser :: Parse Char Expr
parser = litParse 'alt' nameParse 'alt' opExpParse
```

... consists of three parts corresponding to the three sorts of expressions.

Part I: Parsing names of variables

```haskell
nameParse :: Parse Char Expr
nameParse = spot isName 'build' Name
```

```haskell
isName :: Char -> Bool
isName x = ('a' <= x && x <= 'z')
```
A Parser for Expressions (2)

Part II: Parsing (fully bracketed binary) operator expressions

\[
opExpParse = (\text{token } '(' >*>\text{ parser } >*>\text{ spot isOp } >*>\text{ parser } >*>\text{ token }'))'\text{'build'} \text{ makeExpr}
\]

Part III: Parsing literals (numerals)

\[
litParse = ((\text{optional (token } '~')) >*>\text{ (neList (spot isDigit)))'\text{'build'} (\text{charlistToExpr . uncurry (+++))}
\]
A Parser for Expressions (3)

Two further parsers

\[
\text{neList} ::= \text{Parse } a \text{ b } \rightarrow \text{Parse } a \ [\text{b}]
\]

\[
\text{optional} ::= \text{Parse } a \text{ b } \rightarrow \text{Parse } a \ [\text{b}]
\]

such that:

- \text{neList } p \text{ recognizes a non-empty list of the objects which are recognized by } p.
- \text{optional } p \text{ recognizes an object recognized by } p \text{ or succeeds immediately.}

Note that \text{neList} and \text{optional} as well as a number of other supporting functions used such as:

- \text{isOp}
- \text{charlistToExpr}
- ... are yet to be defined (\(\sim\) homework).
The Top-level Parser: Putting it all Together

Converting a string to the expression it represents:

```haskell
topLevel :: Parse a b -> [a] -> b
topLevel p inp
  = case results of
      [] -> error "parse unsuccessful"
      _ -> head results
    where
      results = [ found | (found, []) <- p inp ]
```

It is worth noting:

- The input string is provided by the value of `inp`.
- The parse is successful, if the result contains at least one parse, in which all the input has been read.
Summing up (1)

Parsers of the form:

```haskell
type Parse a b = [a] -> [(b,[a])]
```

```haskell
none :: Parse a b
succeed :: b -> Parse a b
spot :: (a -> Bool) -> Parse a a
alt :: Parse a b -> Parse a b -> Parse a b
>*> :: Parse a b -> Parse a c -> Parse a (b,c)
build :: Parse a b -> (b -> c) -> Parse a c
topLevel :: Parse a b -> [a] -> b
```

...support particularly well the construction of so-called recursive descent parsers.
Summing up (2)

The following language features proved invaluable for combinator parsing:

- **Higher-order functions**: `Parse a b` is of a functional type; all parser combinators are thus higher-order functions, too.

- **Polymorphism**: Consider again the type of `Parse a b`: We do need to be specific about either the input or the output type of the parsers we build. Hence, the above parser combinator can immediately be reused for other (token-) and data types.

- **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 13.1: Further Reading (1)


- Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010. (Chapter 9, Parsing)

Chapter 13.1: Further Reading (2)


Chapter 13.1: Further Reading (3)


Chapter 13.1: Further Reading (4)


Chapter 13.1: Further Reading (5)

(Chapter 17.5, Case study: parsing expressions)

Chapter 13.2

Monadic Parsing
Monadic Parsing

The class Monad

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

The type of a parser function:

```haskell
newtype Parser a = Parser (String -> [(a,String)])
```

We then use the same convention as in Chapter 13.1, i.e.:

- **Delivery of the empty list**: Signals failure of the analysis.
- **Delivery of a non-empty list**: Signals success of the analysis; each element of the list is a pair, whose first component is the identified object (token) and whose second component the input still to be examined.
A Monad of Parsers

Basic Parsers:

- **Recognizing single characters**

```
item :: Parser Char
item = Parser (\cs -> case cs of
  ""    -> []
  (c:cs) -> [(c,cs)]
```

**Note:**

- The functions `item` and `token` are similar.
The Parser Monad (1)

**Parser** is a type constructor. This allows:

```haskell
instance Monad Parser where
    return a = Parser (\cs -> [(a,cs)])
    p >>= f
        = Parser (\cs -> concat [parse (f a) cs' | (a,cs') <- parse p cs])
```

925/1368
The Parser Monad (2)

Note:

- The parser \texttt{return a} succeeds without consuming any of the argument string, and returns the single value \texttt{a}.

- \texttt{parse} denotes a deconstructor function for parsers defined by \texttt{parse (Parser p) = p}.

- The parser \texttt{p >>= f} first applies the parser \texttt{p} to the argument string \texttt{cs} yielding a list of results of the form \texttt{(a,cs')}, where \texttt{a} is a value and \texttt{cs'} is a string. For each such pair \texttt{f a} is a parser that is applied to the string \texttt{cs'}. The result is a list of lists that is then concatenated to give the final list of results.
As required for instances of the class `Monad`, we can show that the 3 monad laws hold:

\[
\begin{align*}
\text{return } a \gg= f &= f \ a \\
p \gg= \text{return} &= p \\
p \gg= (\lambda a \to (f \ a \gg= g)) &= (p \gg= (\lambda a \to f \ a)) \gg= g
\end{align*}
\]
Properties of return and (>>=)

Reminder:

- The above properties are required for each instance of class `Monad`, not just for the specific instance of the parser monad
  - `return` is left-unit and right-unit for `(>>=)`
    - allows a simpler and more concise definition of some parsers.
  - `(>>=)` is associative
    - allows suppression of parentheses when parsers are applied sequentially.
This way we get another two important parsers:

- The always successful parser: `return`
- Sequencing of parsers: `(>>=)`

Note:

The functions

- `return` and `succeed`
- `(>>=)` and `(>>*)`

correspond to each other.
Typical Structure of a Parser (1)

Using the sequencing operator (>>=):

\[ p_1 >>= \lambda a_1 \rightarrow \]
\[ p_2 >>= \lambda a_2 \rightarrow \]
\[ \ldots \]
\[ p_n >>= \lambda a_n \rightarrow \]
\[ f \ a_1 \ a_2 \ldots \ a_n \]
Typical Structure of a Parser (2)

Intuition:

There is a natural operational reading of such a parser:

- Apply parser $p_1$ and denote its result value $a_1$.
- Apply subsequently parser $p_2$ and denote its result value $a_2$.
- ...
- Apply concludingly parser $p_n$ and denote its result value $a_n$.
- Combine finally the intermediate result values by applying some suitable function $f$. 
Typical Structure of a Parser (3)

The do-notation allows a more appealing notation:

```plaintext
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

Alternatively, in just one line:

```plaintext
do {a1 <- p1; a2 <- p2;...; an <- pn; f a1 a2...an}
```
Notational Conventions

Expressions of the form

- $a_i \leftarrow p_i$ are called generators
  (since they generate values for the variables $a_i$)

Remark:

A generator of the form $a_i \leftarrow p_i$ can be
- replaced by $p_i$, if the generated value will not be used afterwards.
Example: A Simple Parser

Write a parser \( p \) that

- reads three characters,
- drops the second character of these, and
- returns the first and the third character as a pair.

Implementation:

\[
p :: \text{Parser} \ (\text{Char},\text{Char})
p \ = \ \text{do} \ \{c \ <- \ \text{item}; \ \text{item}; \ d \ <- \ \text{item}; \ \text{return} \ (c,d)\}
\]
Parser Extensions (1)

Monads with a `zero` and a `plus` are captured by two built-in class definitions in Haskell:

```haskell
class Monad m => MonadZero m where
  zero :: m a

class MonadZero m => MonadPlus m where
  (++) :: m a -> m a -> m a
```

The type constructor \texttt{Parser} can be made instances of these two classes as follows giving two further parsers:

- The parser that always fails:

  ```haskell
  instance MonadZero Parser where
  zero = Parser (\cs -> [])
  ```

- The parser that non-deterministically selects:

  ```haskell
  instance MonadPlus Parser where
  p ++ q = (\cs -> parse p cs ++ parse q cs)
  ```
Simple Properties (1)

We can prove:

\[
\begin{align*}
\text{zero} ++ p &= p \\
 p ++ \text{zero} &= p \\
 p ++ (q ++ r) &= (p ++ q) ++ r
\end{align*}
\]

Informally:

- \text{zero} is left-unit and right-unit for (++)
- (++) is associative

Remark: The above properties are required to hold for each monad with \text{zero} and \text{plus}. 
Specifically for the parser monad we can additionally prove:

\[
\begin{align*}
\text{zero } & \implies f = \text{zero} \\
p & \implies \text{const zero} = \text{zero} \\
(p \mathbin{++} q) & \implies f = (p \implies f) \mathbin{++} (q \implies f) \\
p & \implies (\lambda a \rightarrow f \ a \mathbin{++} g \ a) = (p \implies f) \mathbin{++} (p \implies g)
\end{align*}
\]

Informally:

- zero is left-zero and right-zero element for (\implies)
- (\implies) distributes through (\mathbin{++})
Deterministic Selection

- The parser that deterministically selects:

\[
(\text{+++}) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a
\]

\[
p \text{ +++ } q
= \text{Parser } (\lambda cs \rightarrow \text{case } \text{parse } (p ++ q) \text{ cs of }
  \[
  [] \rightarrow []
  (x:xs) \rightarrow [x]
  \])
\]

It is worth noting:

- (+++\text{)} shows the same behavior as (++\text{)}, but yields at most one result
- (+++\text{)} satisfies all of the previously listed properties of (++\text{)}}
Further Parsers

Recognizing

- **Single objects**
  
  ```haskell
  char :: Char -> Parser Char
  char c = sat (c ==)
  ```

- **Single objects satisfying a particular property**
  
  ```haskell
  sat :: (Char -> Bool) -> Parser Char
  sat p
  = do {c <- item; if p c then return c else zero}
  ```

- **Sequences of numbers, lower case and upper case characters, etc.**
  
  ...analogously to `char`

It is worth noting:

- `sat` and `char` correspond to `spot` and `token`.
Recursion Combinators (1)

Useful parsers can often recursively be defined:

- **Parse a specific string**

  ```haskell
  string :: String -> Parser String
  string "" = return ""
  string (c:cs)
   = do {char c; string cs; return (c:cs)}
  ``

- **Repeated applications of a parser p**

  (Zero or more applications of p)
  ```haskell
  many :: Parser a -> Parser [a]
  many p = many1 p +++ return []
  ``

  (One or more applications of p)
  ```haskell
  many1 :: Parser a -> Parser [a]
  many1 p
   = do a <- p; as <- many p; return (a:as)
  ```
Recursion Combinators (2)

- A variant of the parser `many` with interspersed applications of the 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)
```
Recursion Combinators (3)

- Repeated applications of a parser \( p \), separated by applications of a parser \( op \), whose result value is an operator that is assumed to associate to the left, and which is used to combine the results from the \( p \) parsers

\[
\text{chainl} :: \text{Parser} \ a \to \text{Parser} \ (a \to a \to a) \\
\to a \to \text{Parser} \ a
\]

\[
\text{chainl} \ p \ op \ a = (p \ 'chainl1' \ op) +++ \text{return} \ a
\]

\[
\text{chainl1} :: \text{Parser} \ a \to \text{Parser} \ (a \to a \to a) \\
\to \text{Parser} \ a
\]

\[
p \ 'chainl1' \ op = \text{do} \ \{a <- p; \text{rest} \ a\}
\]

where

\[
\text{rest} \ a = (\text{do} \ f <- op \\
\quad b <- p \\
\quad \text{rest} \ (f \ a \ b))
\]

+++ \text{return} \ a
Lexical Combinators (1)

Suitable combinators allow suppression of a lexical analysis (token recognition), which traditionally precedes parsing:

- Parsing of a string with blanks and line breaks

  \[
  \text{space} ::= \text{Parser String} \\
  \text{space} = \text{many} \ (\text{sat isSpace})
  \]

- Parsing of a token by means of parsers \( p \)

  \[
  \text{token} ::= \text{Parser} \ a \rightarrow \text{Parser} \ a \\
  \text{token} \ p = \text{do} \ \{ \ a \leftarrow \ p; \ \text{space}; \ \text{return} \ a \}
  \]
Lexical Combinators (2)

- Parsing of a symbol token
  \[
  \text{\texttt{symb}} :: \text{String} \rightarrow \text{Parser} \text{ String} \\
  \text{\texttt{symb \ cs}} = \text{token} \ (\text{\texttt{string \ cs}})
  \]

- Application of parser \( p \), removal of initial blanks
  \[
  \text{\texttt{apply}} :: \text{Parser} \ a \rightarrow \text{String} \rightarrow [(a,\text{String})] \\
  \text{\texttt{apply \ p}} = \text{parse} \ (\text{do \ \{space; \ p\}})
  \]
Example: Parsing of Expressions (1)

Grammar:

...for arithmetic expressions built up from single digits using the operators +, −, *, /, and parentheses:

expr ::= expr addop term | term
term ::= term mulop factor | factor
factor ::= digit | (expr)
digit ::= 0 | 1 | ... | 9

addop ::= + | −
mulop ::= * | /
Example: Parsing of Expressions (2)

Parsing and evaluating expressions (yielding integer values) using the \texttt{chainl1} combinator to implement the left-recursive production rules for \texttt{expr} and \texttt{term}:

\begin{verbatim}
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)}
\end{verbatim}
Example: Parsing of Expressions (3)

Example:

Evaluating

apply expr " 1 - 2 * 3 + 4 "

gives the singleton list

[(-1,""')] as desired

as desired.
Chapter 13.2: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer Verlag, 2011. (Kapitel 19.10.5, \( \lambda \)-Parser)


Chapter 13.2: Further Reading (2)


Chapter 13.2: Further Reading (3)


Chapter 14
Logical Programming Functionally
Logical Programming Functionally

Declarative programming

- Functional style
- Logical style

If each of these two styles is appealing

- a combination of (features of) functional and logical programming

should be even more appealing.
Recent Article


...highlights the benefits of combining the paradigm features of both logical and functional programming.

Some of the essence of this article is summarized on the next couple of slides.
Evolution of Programming Languages

...the stepwise introduction of abstractions hiding the underlying computer hardware and the details of program execution.

- **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 Prog. Lang. (Cont’d)

- **Declarative languages**, most prominently **functional** and **logic languages** hide the order of evaluation by removing assignment and other control statements.
  - A declarative program is a set of logical 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 the application components.
- Declarative languages are similar to formal specification languages **but** executable.
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: Examples (1)

- Curry
  http://www.curry-language.org/
  (vers. 0.8.3, September 11, 2012),
  http://www.informatik.uni-kiel.de/~curry/report.html

- TOY
Functional Logic Languages: Examples (2)

- **Mercury**


See also: The Mercury Programming Language http://www.mercurylang.org
And there are many more:

- Escher
- Oz
- HAL
- ...
A Curry Appetizer

Regular Expressions

data RE a = Lit a
  | Alt (RE a) (RE a)
  | Conc (RE a) (RE a)
  | Star (RE a)

The Semantics of Regular Expressions

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))

Note: The Curry-operator ? denotes nondeterministic choice.
A Curry Appetizer (Cont’d)

\[
\text{abstar} = \text{Conc (Lit 'a') (Star (Lit 'b'))}
\]

\[
\text{sem abstar} \rightarrow ["a","ab","abb"]
\]

The Curry-operator \(=:=\) indicates that an equation is to be solved rather than an operation to be defined; here it checks whether a given word \(w\) is in the language of a given regular expression \(re\):

\[
\text{sem re} =:= w
\]

The following equation checks whether a string \(s\) contains a word generated by a regular expression \(re\) (similar to Unix’s grep utility):

\[
xs ++ \text{sem re} ++ ys =:= s
\]

where \(xs, ys\) free
In this chapter

...we will follow a different approach that has been presented in


We will show how to

► integrate features of logical programming into functional programming.

Central means:

► Monads and monadic programming.
Declarative Programming

- **Distinguishing:** Emphasizes the “what”, rather than the “how”
  - **Essence:** Programs are declarative assertions about a problem rather than imperative solution procedures.

- **Variants:** functional and logical programming.

- **Question:** Can functional and logical programming be uniformly combined?
Combining Features of Functional and Logical Programming

Basic approaches:

- **Ambitious**: Designing new programming languages, which enjoy features of both programming styles (e.g. Curry).
- **Simpler**: Implementing an interpreter for one style using the other style.
- **Still simpler**: Write “logical” programs in Haskell using a library of combinators.

⇝ this is the approach taken in the following!
Further Reading

...on functional/logical programming languages:


Remarks on the present Combinator Approach

Advantages and disadvantages

- compared to dedicated functional/logical programming languages
  - less costly
  - but less expressive

Key problems

- Modelling logical programs
  - yielding multiple answers
  - with logical variables (no distinction between input and output variables)
- Modelling the evaluation strategy inherent to logical programs
Running Example: Factoring of Nat. Numbers

Factoring of Natural Numbers: Decomposing a positive integer into the set of pairs of its factors.

Example:

<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>

Obvious Solution:

```hs
factor :: Int -> [(Int,Int)]
factor n = [(r,s) | r<-[1..n], s<-[1..n], r*s == n]
```

In fact, we get:

```hs
?factor 24
[(1,24), (2,12), (3,8), (4,6), (6,4), (8,3), (12,2), (24,1)]
```
Note

The previous solution exploits:

- Explicit domain knowledge
  - E.g. \( r \times s = n \Rightarrow r \leq n \land s \leq n \)
  - This renders possible: Restriction to a finite search space \([1..24] \times [1..24]\)

Often such knowledge is not available; in general:

- The search space cannot be restricted a priori
- In the following thus: Considering the factoring problem as a search problem over an infinite search space \([1..] \times [1..]\)
Tackling the 1st Problem: Several Results

**Solution:** Lists of successes

\[\rightsquigarrow \text{lazy lists} \ (\text{Phil Wadler})\]

**Idea**

- A function of type \(a \rightarrow b\) can be replaced by a function of type \(a \rightarrow [b]\).

- **Lazy evaluation** ensures that the elements of the result list (list of successes) are provided as they are found, rather than as a complete list after termination of the computation.
Realizing this idea in the factoring example (assuming that the search space cannot be bounded a priori):

```plaintext
factor :: Int -> [(Int,Int)]
factor n = [(r,s) | r<-[1..], s<-[1..], r*s == n]

?factor 24
[(1,24)]

...followed by an infinite wait.

⇝ This is of no practical value!
Remedy: Fair Order via Diagonalization (1)

Explore the search space of pairs in a fair order:

\[
\text{factor } n = \{(r,s) \mid (r,s) \leftarrow \text{diagprod } [1..][1..], \ r \times s = n\}
\]

where

\[
\text{diagprod} :: [a] \rightarrow [b] \rightarrow [(a,b)]
\]
\[
\text{diagprod} \ x s \ y s = \{(x s!!i, \ y s!!(n-i)) \mid n \leftarrow [0..], \ i \leftarrow [0..n]\}
\]

Effect: Each pair \( (x, y) \) is now reached after a finite number of steps:

\[
\{(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3), (3,2), \ldots \}
\]
Remedy: Fair Order via Diagonalization (2)

Applied to our running example, we obtain:

```
?factor 24
[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)
```

...this means, we obtain all results; followed again, however, by an infinite wait.

Of course, this was expected, since the search space is infinite.
Systematic Remedy: Using Monads

Reminder:

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Notational conventions for the following development:

- **Stream a** ...for (potentially) infinite lists
- **[a]** ...for finite lists
- **Note:** The distinction between **Stream a** for infinite lists and **[a]** for finite lists is only conceptually; the following definition makes this explicit:

  ```haskell
type Stream a = [a]
```
List Monad

The monad of (potentially infinite) lists

Definition of the monad operations

- **return** (yields the singleton list):
  
  return :: a -> Stream a
  return x = [x]

- binding operator (**>>=**):
  
  (>>=) :: Stream a -> (a -> Stream b) -> Stream b
  xs >>= f = concat (map f xs)

Other monad operations are irrelevant in our context.
Benefit

The monad operations `return` and `(>>=)` allow us to model/replace list comprehension:

The meaning of the expression, for example,

\[(x, y) \mid x \leftarrow [1..], y \leftarrow [10..]\]

...using list comprehension is equivalent to

\[\text{concat } \left( \text{map } (\lambda x \rightarrow [(x, y) \mid y \leftarrow [10..]]) [1..] \right)\]

...that itself is equivalent to

\[\text{concat } \left( \text{map } (\lambda x \rightarrow \text{concat } \left( \text{map } (\lambda y \rightarrow [(x, y)]) [10..])) [1..] \right)\]

Using `return` and `(>>=)` this can concisely be expressed by:

\[[1..] >>= (\lambda x \rightarrow [10..] >>= (\lambda y \rightarrow \text{return } (x, y)))\]
Benefit (Cont’d)

Haskell’s do-notation allows an even more compact equivalent representation:

```haskell
do x <- [1..]; y <- [10..]; return (x,y)
```

Recalling the general rule:

The expression

```haskell
do x1 <- e1; x2 <- e2; ... ; xn <- en; e
```

is a shorthand for

```haskell
e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```
Fairness: Adapting the binding op. (>>=) (1)

Are we done? Not yet, since:

- Exploring the pairs of the search space is still not fair.

The expression

```haskell
do x <- [1..]; y <- [10..]; return (x,y)
```

yields the stream

```
[(1,10), (1,11), (1,12), (1,13), (1,14), ...
```

This problem is going to be tackled next.
Fairness: Adapting the binding op. (>>=) (2)

Idea: Embedding diagonalization into (>>=)

Implementation

Introducing a new type \textbf{Diag} a:

\begin{verbatim}
newtype Diag a = MkDiag (Stream a) deriving Show
\end{verbatim}

...together with an auxiliary function for stripping off the type constructor \textbf{MkDiag}:

\begin{verbatim}
unDiag (MkDiag xs) = xs
\end{verbatim}
Fairness: Adapting the binding op. (\(\gg=\)) (3)

Diag is made an instance of the constructor class Monad:

\[
\text{instance Monad Diag where}
\]
\[
\text{return } x = \text{MkDiag } [x]
\]
\[
\text{MkDiag } xs \gg= f
\]
\[
= \text{MkDiag } (\text{concat } (\text{diag } (\text{map } (\text{unDiag } . f) xs)))
\]

where \text{diag} rearranges the values into a fair order:

\[
\text{diag} :: \text{Stream (Stream } a) \rightarrow \text{Stream } [a]
\]
\[
\text{diag } [] = []
\]
\[
\text{diag } (xs:xss)
\]
\[
= \text{lzw } (++) \left[ \left[ x \right] \mid x \leftarrow xs \right] ([] : \text{diag } xss)
\]
Fairness: Adapting the binding op. ( >>= ) (4)

where

\[
\text{lzw} :: (a \to a \to a) \to \text{Stream } a \to \text{Stream } a \to \text{Stream } a
\]

\[
lzw \ f \ [] \ ys = ys
\]

\[
lzw \ f \ xs \ [] = xs
\]

\[
lzw \ f \ (x:xs) \ (y:ys) = (f \ x \ y) : (lzw \ f \ xs \ ys)
\]

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.
Fairness: Adapting the binding op. ( >>= ) (5)

Intuition:

- `return` yields the singleton list.
- `undiaq` strips off the constructor added by the function
  \[ f :: a \to Diag b. \]
- `diag` arranges the elements of the list into a fair order
  (and works equally well for finite and infinite lists).
- `lzw` reminds to “like zipWith.”
Fairness: Adapting the binding op. (>>=) (6)

The idea underlying `diag`:

- Transform an infinite list of infinite lists
  
  \[
  [[[x_{11}, x_{12}, x_{13}, \ldots]], [[x_{21}, x_{22}, \ldots]], [[x_{31}, x_{32}, \ldots]], \ldots]
  \]

- ...into an infinite list of finite diagonals
  
  \[
  [[[x_{11}], [x_{12}, x_{21}], [x_{13}, x_{22}, x_{31}], \ldots]]
  \]

This way:

```
?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),..]
```

Summing up

- We have achieved: The pairs are delivered in a fair order!
Back to the Factoring Problem (1)

Current state of our solution:

- Generating pairs (in a fair order): done.
- Selecting (those pairs being part of the solution): still open.

Approach for solving the selection problem, i.e., filtering out the pairs \((r, s)\) satisfying the equality \(r \times s = n\):

- Filtering with conditions!
Back to the Factoring Problem (2)

For that purpose:

```haskell
class Monad m => Bunch m where
  zero :: m a  -- empty result, no answer
  alt :: m a -> m a -> m a  -- all answers either
                              -- in xm or ym
  wrap :: m a -> m a  -- answers yielded by auxiliary calculations; right
                     -- now, wrap is defined as
                     -- the identity function
```

Note:

▷ The value `zero` allows to express an empty answer set.
Back to the Factoring Problem (3)

In detail:

The instance declaration for ordinary lazy lists:

```haskell
instance Bunch [] where
  zero = []
  alt xs ys = xs ++ ys
  wrap xs = xs
```

...and for the monad `Diag`:

```haskell
instance Bunch Diag where
  zero = MkDiag[]
  alt (MkDiag xs)(MkDiag ys) = MkDiag (shuffle xs ys)
  wrap xm = xm

shuffle [] ys = ys
shuffle (x:xs) ys = x : shuffle ys xs
```

(Remark: `alt` and `wrap` will be used later.)
Back to the Factoring Problem (4)

By means of \texttt{zero}, the function \texttt{test} yields the key for filtering:

\begin{verbatim}
test :: Bunch m => Bool -> m()
test b = if b then return() else zero
\end{verbatim}

This does not look useful, but it provides the key to filtering:

\begin{verbatim}
?do x <- [1..]; () <- test (x 'mod' 3 == 0); return x
[3,6,9,12,15,18,21,24,27,30,33,..
\end{verbatim}

\begin{verbatim}
?do x <- MkDiag [1..]; test (x 'mod' 3 == 0); return x
MkDiag[3,6,9,12,15,18,21,24,27,30,33,..
\end{verbatim}
Are we done? (1)

Not yet!

Consider:

?do r <- MkDiag[1..]; s <- MkDiag[1..];
   test(r*s==24); return (r,s)
MkDiag[(1,24)

...followed by an infinite wait.

What are the reasons for that?

do r <- MkDiag[1..]; s <- MkDiag[1..];
   test(r*s==24); return (r,s)

is equivalent to

do x <- MkDiag[1..]
   (do y <- MkDiag[1..]; test(x*y==24);
    return (x,y))
Are we done? (2)

I.e., the generator for $y$ is merged with the subsequent test to the following (sub-) expression:

$$\text{do } y \leftarrow \text{MkDiag}[1..]; \text{test}(x\cdot y==24); \text{return } (x,y)$$

**Intuition:**

- This expression yields for a given value of $x$ all values of $y$ with $x \cdot y = 24$.
- For $x = 1$ the answer $(1, 24)$ will be found, in order to search in vain for further values of $y$.
- For $x = 5$ we thus do not observe any output.
Solution Approach

The deeper reason for this undesired behaviour:

The missing associativity of (\ >>=) for Diag, i.e.,

\[ (x \ >>= f) \ >>= g = x \ >>= (\lambda x \rightarrow f x \ >>= g) \]

...does not hold for (\ >>=) and Diag!

Remedy: Explicit grouping of generators to ensure fairness

?do (x,y) <- (do u <- MkDiag[1..];
    v <- MkDiag[1..]; return (u,v))
    test (x*y==24); return (x,y)
MkDiag[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),
       (1,24),(24,1)]

...all results, subsequently followed by an infinite wait.
Remarks

- All results, subsequently followed by an infinite wait
  ↞ this is the best we can hope for if the search space is infinite.

- Explicit grouping
  ↞ required only because of missing associativity of (>>=), otherwise both expressions would be equivalent.

- In the following
  ↞ avoid infinite waiting by indicating that a result has not (yet) been found.
Indicating that no solution is found

To this purpose: Introducing a new type Matrix together with breadth search.

Intuition

▶ Type Matrix: Infinite list of finite lists.
▶ Goal: A program that yields a matrix of answers, where row \( i \) contains all answers that can be computed with costs \( c(i) \).
▶ Solving the indication problem: By returning the empty list in a row (means “nothing found”).
Implementation (1)

The new type Matrix

newtype Matrix a  
  = MkMatrix (Stream [a]) deriving Show

...with an auxiliary function for stripping off the constructor:

unMatrix (MkMatrix xm) = xm
Implementation (2)

Preliminary definitions for making \texttt{Matrix} an instance of class \texttt{Bunch}:

\begin{verbatim}
return x = MkMatrix[[x]]  -- Matrix with a single row
zero = MkMatrix[]          -- Matrix without rows
alt(MkMatrix xm) (MkMatrix ym) = MkMatrix(lzw (+) xm ym)
wrap(MkMatrix xm) = MkMatrix([],xm)  -- the clou is encoded in wrap!

(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))
bindm :: Stream[a] -> (a -> Stream[b]) -> Stream[b]
bindm xm f = map concat (diag (map (concatAll . map f) xm))
concatAll :: [Stream [b]] -> Stream [b]
concatAll = foldr (lzw (+)) []
\end{verbatim}
Implementation (3)

In total we are now ready to make Matrix an instance of the classes Monad and Bunch:

```haskell
instance Monad Matrix where
    return x            = MkMatrix[[x]]
    (MkMatrix xm) >>= f = MkMatrix(bindm xm (unMatrix . f))

instance Bunch Matrix where
    zero                     = MkMatrix[]
    alt(MkMatrix xm)(MkMatrix ym) = MkMatrix(lzw (++) xm ym)
    wrap(MkMatrix xm)        = MkMatrix([]:xm)

intMat = MkMatrix[[n] | n <- [1..]]
```

Example:

```haskell
?do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
MkMatrix[[[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [(4,6),(6,4)],
           [(3,8),(8,3)], [[]], [[]], [(2,12),(12,2)], [[]], [[]], [[]],
           [[]], [[]], [[]], [[]], [[], [], [], [], [(1,24),(24,1)], [[]], [[]], []], ..
```
A Variety of Search Strategies

(i) Breadth search ($\text{MkMatrix}[[n] \mid n<-[1..]]$), (ii) depth search ($[1..]$), (iii) diagonalization:

...by means of additional functions that allow us to fix the strategy of interest at the time of calling ("just in time").

Control via a monad type:

```haskell
choose :: Bunch m => Stream a -> m a
choose (x:xs) = wrap (return x 'alt' choose xs)

factor :: Bunch m => Int -> m(Int, Int)
factor n = do r <- choose[1..]; s <- choose[1..];
             test(r*s==n); return (r,s)
```
Selecting a Search Strategy

This allows:

- Usage of `factor` with different search strategies.
- The specified type of `factor` determines the search monad (and thus the search strategy).

```haskell
?factor 24 :: Stream(Int,Int)
[(1,24)]

?factor 24 :: Matrix(Int, Int)
Matrix[[],[],[],[],[],[],[],[],[],[],[(4,6),(6,4)],
[(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],[],[],
[],[],[],[],[],[],[(1,24),(24,1)],[],[],[]]..
Summary of Progress

Recall:

The 3 key problems we had/have to deal with:

▶ Modelling logical programs with
  ▶ multiple results: done (essentially by means of lazy lists)
  ▶ logical variables: still open
    ▶ Common for logical 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.
  ▶ Modelling of the evaluation strategy inherent to logical programs: done
    ▶ implicit search of logical programming languages has been made explicit.
    ▶ by means of type classes of Haskell even different search strategies were conveniently be realizable.
Tackling the Final Problem: Terms, Substitutions & Predicates (1)

Towards the modeling in Haskell:

Terms will describe values of logical variables:

```haskell
data Term = Int Int
           | Nil
           | Cons Term Term
           | Var Variable deriving Eq
```

Named variables will be used for formulating queries, generated variables evolve in the course of the computation:

```haskell
data Variable = Named String
               | Generated Int deriving (Show, Eq)
```
Terms, Substitutions & Predicates (2)

Some auxiliary functions

- for transforming a string into a named variable
  
  \[
  \text{var} :: \text{String} \rightarrow \text{Term} \\
  \text{var} \ s = \ \text{Var} \ (\text{Named} \ s)
  \]

- for constructing a term representation of a list of integers
  
  \[
  \text{list} :: [\text{Int}] \rightarrow \text{Term} \\
  \text{list} \ \text{xs} = \ \text{foldr} \ \text{Cons} \ \text{Nil} \ (\text{map} \ \text{Int} \ \text{xs})
  \]
Substitution and unification:

-- Substitution is essentially a mapping
-- from variables to terms (details later)

```
newtype Subst
```

Further support functions:

```
apply :: Subst -> Term -> Term
idsubst :: Subst
unify :: (Term, Term) -> Subst -> Maybe Subst
```
Logical programs (in our Haskell environment) with \( m \) of type bunch:

```
-- Logical programs have type Pred m
type Pred m = Answer -> m Answer

-- Answers; the integer-component controls
-- the generation of new variables
newtype Answer = MkAnswer (Subst, Int)
```
Terms, Substitutions & Predicates (5)

```
-- "Initial answer"
initial :: Answer
initial = MkAnswer (idsubst, 0)
run :: Bunch m => Pred m -> m Answer
run p = p initial

-- "Program run of a predicate as query", where
-- p is applied to the initial answer
run p :: Stream Answer
```
Writing logical programs

Example:

\texttt{append(a,b,c)} where \texttt{a}, \texttt{b} denote lists and \texttt{c} the concatenation of the lists \texttt{a} and \texttt{b}.

Implementation as a function of terms on predicates:

\texttt{append :: Bunch m => (Term, Term, Term) \rightarrow Pred m}

\texttt{\begin{verbatim}
-- The implementation of append (will follow!) and
-- of appropriate Show-Functions is supposed:
?run(append(list[1,2],list[3,4],var "z"))
\end{verbatim}}

\texttt{:: Stream Answer}

\texttt{[z=[1,2,3,4]]}

\texttt{\begin{verbatim}
-- Note: Equivalent to the above list but more
-- accurate would be:
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
\end{verbatim}}
Combinators for logical programs (1)

Simple predicates are formed by means of the operators (=:=) (equality of terms):

?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)) =
  case unify(tu) s of
  Just s’ -> return(MkAnswer(s’,n))
  Nothing -> zero
Combinators for logical programs (2)

Conjunction of predicates by means of the operator (&&&) (conjunction):

\[ \text{?run(var "x" =:= Int 3 &&& var "y" =:= Int 4)} \]
\[ \text{:: Stream Answer} \]
\[ [{x=3,y=4}] \]

\[ \text{?run(var "x" =:= Int 3 &&& var "x" =:= Int 4)} \]
\[ \text{:: Stream Answer} \]
\[ [] \]

Implementation by means of the operator (>>=) of type bunch:

\[ (&&&) :: \text{Bunch m} \to \text{Pred m} \to \text{Pred m} \to \text{Pred m} \]
\[ (p &&& q) s = p s >>= q \]

-- equivalent and highlighting the
-- sequentiality would be
do t <- p s; u <- q t; return u
Combinators for logical programs (3)

Disjunction of predicates by means of the operator $\texttt{(|||)}$ (Disjunction):

\[
\texttt{?run(var "x" == Int 3 ||| var "x" == Int 4)} \\
\texttt{: Stream Answer} \\
\texttt{[{x=3,x=4}]} \\
\]

Implementation by means of the operator $\texttt{alt}$ of type $\texttt{bunch}$:

\[
\texttt{(|||) :: Bunch m -> Pred m -> Pred m -> Pred m} \\
\texttt{(p ||| q) s = alt (p s) (q s)} \\
\]
Combinators for logical programs (4)

Introducing new variables in predicates (exploiting the integer-component of answers)

...on the construction of local variables in recursive predicates:

\[
\text{exists} :: \text{Bunch m} \Rightarrow (\text{Term} \rightarrow \text{Pred m}) \rightarrow \text{Pred m}
\]

\[
\text{exists} \; p \; (\text{MkAnswer} \; (s,n)) = \\
\quad p \; (\text{Var(Generated n)}) \; (\text{MkAnswer}(s,n+1))
\]

Also for handling recursive predicates

... ensures that in connection with \text{Matrix} the costs per recursion unfolding increase by 1:

\[
\text{step} :: \text{Bunch m} \Rightarrow \text{Pred m} \rightarrow \text{Pred m}
\]

\[
\text{step} \; p \; s = \; \text{wrap} \; (p \; s)
\]
Example

Examples of applications of `wrap` and `step`:

```plaintext
?run (var "x" =:= Int 0) :: Matrix Answer
MkMatrix[[] , [{x=0}]]

?run(step(var "x" =:= Int 0)) :: Matrix Answer
MkMatrix[[] , [{x=0}]]
```
Recursive Programs (1)

This allows us to provide the implementation of `append`:

```haskell
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)))))
```

As common for logical programs, also the following application of `append` is possible:

- The concatenation of which lists equals the list `[1,2,3]`?

```erlang
?run(append(var "x", var "y", list[1,2,3]))  :: Stream Answer

[{x = Nil, y = [1,2,3]},
 {x = [1], y = [2,3]},
 {x = [1,2], y = [3]},
 {x = [1,2,3], y = Nil}]
```
A More Complex Example (1)

Constructing “good” sequences consisting of 0s and 1s.

Definition:

1. The sequence $[0]$ is good.
2. If the sequences $s_1$ and $s_2$ are good, then also the sequence $[1] ++ s_1 ++ s_2$.
3. Except of the sequences according to 1. and 2., there are no other good sequences.
A More Complex Example (2)

Implementation as a predicate:

good(s) =
   step (s =:= Cons(Int 0) Nil
   ||| exist (\t -> exists (\q -> exists (\r ->
       s =:= Cons (Int 1) t &&& append(q,r,t)
       &&& good(q) &&& good(r))))))
Applications (1)

1) Test of being “good”:

?run (good (list[1,0,1,1,0,0,1,0,0]))
   :: Stream Answer
[{}] -- empty answer set, if argument list is good

?run (good (list[1,0,1,1,0,0,1,0,1]))
   :: Stream Answer
[] -- no answer, if argument list is not good

Note:

▶ “empty answer” and “no answer” correspond to “yes” and “no” of a Prolog system.
Applications (2)

2) Constructing “good” lists

-- With an unfair bunch-type: Some answers are missing

?run(good(var "s")) :: Stream Answer

[s=[0],
  s=[1,0,0],
  s=[1,0,1,0,0],
  s=[1,0,1,0,1,0,0],
  s=[1,0,1,0,1,0,1,0,0],...

Applications (3)

-- For comparison: With a fair bunch-type
?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],
    s=[1,0,1,1,1,0,0],
    s=[1,1,0,0,1,0,1,0,0],
    s=[1,1,0,0,1,0,1,0,0],
    s=[1,1,0,0,1,0,1,0,0],
    s=[1,1,0,0,1,0,1,0,0],
    s=[1,1,0,0,1,0,1,0,0],...
Applications (4)

-- For comparison: With a breadth-first search
-- bunch-type. Effect: The output of results is
-- more "predictable".

?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]],[],
..
Delivering Missing Definitions (1)

Priorities of new infix operators:

```
infixr 4  :=:
infixr 3  &&&
infixr 2  |||
```

Substitution:

```
newtype Subst = MkSubst [(Var, Term)]
unSubst(MkSubst s) = s

idsubst = MkSubst[]
extend x t (MkSubst s) = MkSubst ((x,t):s)
```
Delivering Missing Definitions (2)

Application of substitution:

apply :: Subst -> Term -> Term
apply s t =
  case deref s t of
    Cons x xs -> Cons (apply s x) (apply s xs)
    t'         -> t'

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
Delivering Missing Definitions (3)

Unification:

\[
\text{unify :: (Term, Term) -> Subst -> Maybe Subst}
\]

\[
\text{unify (t,u) s =}
\]

\[
\text{case (deref s t, deref s u) of}
\]

\[
\text{\hspace{1cm} (Nil, Nil) -> Just s}
\]

\[
\text{\hspace{1cm} (Cons x xs, Cons y ys) -> unify (x,y) s >>= unify (xs, ys)}
\]

\[
\text{\hspace{1cm} (Int n, Int m) | (n==m) -> Just s}
\]

\[
\text{\hspace{1cm} (Var x, Var y) | (x==y) -> Just s}
\]

\[
\text{\hspace{1cm} (Var x, t) \rightarrow if occurs x t s}
\]

\[
\text{\hspace{3cm} then Nothing}
\]

\[
\text{\hspace{3cm} else Just (extend x t s)}
\]

\[
\text{\hspace{1cm} (t, Var x) \rightarrow if occurs x t s}
\]

\[
\text{\hspace{3cm} then Nothing}
\]

\[
\text{\hspace{3cm} else Just (extend x t s)}
\]

\[
\text{\hspace{1cm} (_,_) \rightarrow Nothing}
\]
Delivering Missing Definitions (4)

occurs :: Variable -> Term -> Subst -> Bool
occurs x t s =
    case deref s t of
        Var y       -> x == y
        Cons y ys   -> occurs x y s || occurs x ys s
        _           -> False
Summing up

Current functional logic languages aim at balancing

▶ generality (in terms of paradigm integration)
▶ efficient 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

▶ an emerging paradigm with appealing features
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 15
Pretty Printing
Motivation

Pretty Printing is

- like lexical and syntactical analysis another typical application used for demonstrating the elegance of functional programming.
What’s it all about?

A pretty-printer is

- a tool (often a library of routines) designed for converting a tree into plain text.

Essential goal of pretty printing:

- Preserving and reflecting the structure of the tree by indentation while using a minimum number of lines.

Hence

- Pretty printing can be considered the converse problem to parsing.
A “Good” Pretty-Printer

...is distinguished by properly balancing

- Simplicity of usage
- Flexibility of the format
- “Prettiness” of output
The presentation in this chapter

...is based on:


It shall improve (see end of chapter) on the below pretty printer library proposed by John Hughes that is widely recognized as a standard:

A Simple Pretty Printer: Basic Approach

**Requirement:** For each document there shall be only one possible layout (e.g., no attempt is made to compress structure onto a single line).

The basic operators needed are:

\[
\begin{align*}
\text{(<>)} & : \text{Doc} \to \text{Doc} \to \text{Doc} & \quad \text{-- associative concatenation of documents} \\
\text{nil} & : \text{Doc} & \quad \text{-- The empty document: Right and left unit for (<>)} \\
\text{text} & : \text{String} \to \text{Doc} & \quad \text{-- Conversion function: Converts a string to a document} \\
\text{line} & : \text{Doc} & \quad \text{-- Line break} \\
\text{nest} & : \text{Int} \to \text{Doc} \to \text{Doc} & \quad \text{-- Adding indentation} \\
\text{layout} & : \text{Doc} \to \text{String} & \quad \text{-- Output: Converts a document to a string}
\end{align*}
\]
Convention

- Arguments of text are free of newline characters.
A Simple Implementation

Implement

- **doc** as strings (i.e. as data type **String**)

with

- **(<>)** as concatenation of strings
- **nil** as empty string
- **text** as identity on strings
- **line** as new line
- **nest i** as indentation: adding *i* spaces (after each line break by means of **line**) → essential difference to Hughes’ pretty printer that also allows inserting spaces in front of strings allowing here to drop one concatenation operator
- **layout** as identity on strings
Example

Converting trees into documents (here: Strings) which are output as text (here: Strings).

Consider the following type of trees:

data Tree = Node String [Tree]

A concrete value B of type Tree:

Node "aaa" [Node "bbbb" [Node "cc" [], Node "dd" []],
               Node "eee" [],
               Node "ffff" [Node "gg" [],
                            Node "hhh" [],
                            Node "ii" []
                     ]
       ]
...and its desired output

A text, where indentation reflects the structure of tree B:

```
  aaa[bbbb[ccc,
        dd],
       eee,
       ffff[gg,
            hhh,
            ii]]
```

It is worth noting:

- **Sibling trees** start on a new line, properly indented.
### Implementation

The below implementation achieves this:

```haskell
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
```
Another possibly wanted output of B

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

**Note:**
- each subtree starts on a new line, properly indented.
An implementation producing the latter output

```haskell
data Tree = Node String [Tree]

showTree' :: Tree -> Doc
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' :: [Tree] -> Doc
showBracket' [] = nil
showBracket' ts = text "[" <> nest 2 (line <>
    showTrees' ts) <> line <> text "]"

showTrees' :: [Tree] -> Doc
showTrees' [t] = showTree t
showTrees' (t:ts) = showTree t <> text "," <> line
    <> showTrees ts
```
Normal Form of Documents

Documents can always be reduced to normal form.

Normal form

- Text alternating with line breaks nested to a given indentation:

  text s0 <> nest i1 line <> text s1 <> ...
  <> nest ik line <> text sk

where

- each $s_j$ is a (possibly empty) string
- each $i_j$ is a (possibly zero) natural number
Example on Normal Forms 1(2)

A document

text "bbbbb" <> text "[" <>
nest 2 (  
   line <> text "ccc" <> text "," <>
   line <> text "dd"
),
line <> text "]"

...and how it is output:

bbbbb[
   ccc,
   dd
]

Example on Normal Forms 2(2)

The same document

text "bbbbb" <> text "[" <>
nest 2 (  
  line <> text "ccc" <> text "," <>
  line <> text "dd"
)

line <> text "]"

...and its normal form:

text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
Why does it work?

...because of the properties (laws) the functions enjoy.

In more detail

...because of the fact that

- $(<>)$ is associative with unit $\text{nil}$
- the laws summarized on the next slide.

Note:

- All of these laws except of the last one are paired; they are paired with a corresponding law for their units.


Properties of the Functions/Laws 1(2)

We have the following (pairs of) laws (except for the last one):

\[
\begin{align*}
text (s \!+\! \ t) & \quad = \quad text \ s \ <> \ text \ t \quad -- \text{text is a homomorphism from string concatenation to document concatenation} \\
text "" & \quad = \quad \text{nil} \\
\end{align*}
\]

\[
\begin{align*}
nest (i\!+\!j) \ x & \quad = \quad nest \ i \ (nest \ j \ x) \quad -- \text{nest is a homomorphism from addition to composition} \\
nest \ 0 \ x & \quad = \quad x \\
nest \ i \ nil & \quad = \quad \text{nil} \\
nest \ i \ (x \ <> \ y) & \quad = \quad nest \ i \ x \ <> \ nest \ i \ y \quad -- \text{nest distributes through document concatenation} \\
nest \ i \ (text \ s) & \quad = \quad text \ s \quad -- \text{Nesting is absorbed by text; different to Hughes’ pretty printer} \\
\end{align*}
\]
Properties of the Functions/Laws 2(2)

Relevance and Impact

- The above laws are sufficient to ensure that documents can always be transformed into normal form
  - First four laws: applied from left to right
  - Last three laws: applied from right to left
Further Properties/Laws

...that relate documents to their layouts:

\[
\begin{align*}
\text{layout} (x <> y) &= \text{layout} x ++ \text{layout} y \\
\text{layout} \text{ nil} &= "" \quad -- \text{layout is a homomorphism} \\
&\hspace{1cm} -- \text{from document concatenation} \\
&\hspace{1cm} -- \text{to string concatenation} \\
\text{layout} (\text{text} s) &= s \quad -- \text{layout is the inverse} \\
&\hspace{1cm} -- \text{of text} \\
\text{layout} (\text{nest} i \ \text{line}) &= '\n' : \text{copy} i \ '' \quad -- \text{layout of a nested line} \\
&\hspace{1cm} -- \text{is a newline followed by} \\
&\hspace{1cm} -- \text{one space for each level} \\
&\hspace{1cm} -- \text{of indentation}
\end{align*}
\]
The Implementation of Doc

Intuition

- Represent documents as a concatenation of items, where each item is a text or a line break indented to a given amount.

This is realized as a sum type (the algebra of documents):

```haskell
data Doc = Nil
         | String 'Text' Doc
         | Int 'Line' Doc
```

The constructors relate to the document operators as follows:

```haskell
Nil = nil
s 'Text' x = text s <> x
i 'Line' x = nest i line <> x
```
Example

Using this new algebraic type Doc, the normal form (considered previously)

text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"

...is represented by the following value of Doc:

"bbbbb["  'Text' (  
2  'Line' ("ccc,"  'Text' (  
2  'Line' ("dd,"  'Text' (  
0  'Line' ("],"  'Text' Nil))))))

Derived Implementations 1(2)

Implementations of the document operators can easily be derived from the above equations:

\[
\begin{align*}
\text{nil} & = \text{Nil} \\
\text{text } s & = s \ 'Text' \ \text{Nil} \\
\text{line} & = 0 \ 'Line' \ \text{Nil} \\
(s \ 'Text' \ x) & <> y = s \ 'Text' \ (x <> y) \\
(i \ 'Line' \ x) & <> y = i \ 'Line' \ (x <> y) \\
\text{Nil} & <> y = y
\end{align*}
\]
Derived Implementations 2(2)

\[
\begin{align*}
nest \ i \ (s \ 'Text' \ x) &= s \ 'Text' \ nest \ i \ x \\
nest \ i \ (j \ 'Line' \ x) &= (i+j) \ 'Line' \ nest \ i \ x \\
nest \ i \ Nil &= Nil \\
\end{align*}
\]

\[
\begin{align*}
layout \ (s \ 'Text' \ x) &= s \ + \ layout \ x \\
layout \ (i \ 'Line' \ x) &= '\n' : \ copy \ i \ ' ' \ + \ layout \ x \\
layout \ Nil &= ""
\end{align*}
\]
Correctness of the derived Implementations

...can be shown for each of them, e.g.:

- Derivation of

  \[(s \ 'Text' \ x) <> y = s \ 'Text' \ (x <> y)\]

  \[(s \ 'Text' \ x) <> y = \{ \text{Definition of Text} \} \]

    \(\text{text} \ s <> x) <> y\]

  \[= \{ \text{Associativity of} \ <> \} \]

    \(\text{text} \ s <> (x <> y)\]

  \[= \{ \text{Definition of Text} \} \]

    \(s \ 'Text' \ (x <> y)\]

The remaining equations can be shown using similar reasoning.
Documents with Multiple Layouts

Adding Flexibility:

- **Up to now:** Documents were equivalent to a string (i.e., they have a fixed single layout)
- **Next:** Documents shall be equivalent to a set of strings (i.e., they may have multiple layouts) where each string corresponds to a layout.

This can be rendered possible by just adding a new function:

\[
group :: \text{Doc} \rightarrow \text{Doc}
\]

**Informally:**

Given a document, representing a set of layouts, `group` returns the set with one new element added that represents the layout in which everything is compressed on one line: Replace each newline (plus indentation) by a single space.
Preferred Layouts

“Beauty” needs to be specified/defined:

- `pretty` replaces `layout`
  
  `pretty :: Int -> Doc -> String`

  and picks the prettiest layout depending on the preferred maximum line width argument.

  **Remark:** `pretty`’s integer-argument specifies the preferred maximum line length of the output (and hence the prettiest layout out of the set of alternatives at hand).
Example

Using the modified `showTree` function based on `group`:

```haskell
showTree (Node s ts) = group (text s <> nest (length s) (showBracket ts))
```

...the call of `pretty 30` (once completely specified) will yield the output:

```text
aaa[bbbb[ccc, dd],
   eee,
   ffff[gg, hhh, ii]]
```

This ensures:

- Trees are fit onto one line where possible (i.e., length \( \leq 30 \)).
- Insertion of sufficiently many line breaks in order to avoid exceeding the given maximum line length.
Implementation of the new Functions

The following supporting functions are required:

-- Forming the union of two sets of layouts
(<>): Doc -> Doc -> Doc

-- Replacement of each line break (and its
-- associated indentation) by a single space
flatten :: Doc -> Doc
Implementation of the new Functions (Cont’d)

▶ **Observation**: A document always represents a non-empty set of layouts.

▶ **Requirements**:
  
  ▶ In \((x <|> y)\) all layouts of \(x\) and \(y\) enjoy the same flat layout (mandatory invariant of \(<|>\)).
  
  ▶ Each first line in \(x\) is at least as long as each first line in \(y\) (second invariant).

▶ **Note**: \(<|>\) and *flatten* are not directly exposed to the user (only via group and other supporting functions).
Properties/Laws of ( <|> )

Operators on simple documents are extended pointwise through union:

\[(x <|> y) <|> z = (x <|> z) <|> (y <|> z)\]
\[x <|> (y <|> z) = (x <|> y) <|> (x <|> z)\]
\[nest \, i \,(x <|> y) = nest \, i \, x <|> nest \, i \, y\]
Properties/Laws of **flatten**

The interaction of **flatten** with other document operators:

\[
\text{flatten} \ (x \ <|> \ y) = \text{flatten} \ x \quad \text{-- distribution law}
\]

\[
\text{flatten} \ (x \ <> \ y) = \text{flatten} \ x \ <> \ \text{flatten} \ y
\]

\[
\text{flatten} \ \text{nil} = \text{nil}
\]

\[
\text{flatten} \ (\text{text} \ s) = \text{text} \ s
\]

\[
\text{flatten} \ \text{line} = \text{text} \ " \ " \quad \text{-- the most interesting case: line breaks are replaced}
\]

\[
\qquad \text{-- by a single space}
\]

\[
\text{flatten} \ (\text{nest} \ i \ x) = \text{flatten} \ x
\]
Implementation of group

...by means of flatten and (<>), the implementation of group can be given:

\[
group x = \text{flatten } x \langle|\rangle x
\]

Intuitively: group adds the flattened layout to a set of layouts.

Note: A document always represents a non-empty set of layouts where all layouts in the set flatten to the same layout.
Normal Form

Based on the previous laws each document can be reduced to a normal form of the form

\[ x_1 <|> \ldots <|> x_n \]

where each \( x_i \) is in the normal form of simple documents (which was introduced previously).
Selecting a “best” Layout out of a Set of Layouts

...by defining an ordering relation on lines in dependence of the given maximum line length.

Out of two lines

- which do not exceed the maximum length, select the longer one
- of which at least one exceeds the maximum length, select the shorter one

Note: Sometimes we have to pick a layout where some line exceeds the limit (a key difference to the approach of Hughes). However, this is done only, if this is unavoidable.
The Adapted Implementation of Doc

The new implementation of \texttt{Doc} as algebraic type. It is similar to the previous one except for the new construct representing the \textit{union of two documents}:

\begin{verbatim}
data Doc = -- As before: The first 3 alternatives
    Nil
  | String 'Text' Doc
  | Int 'Line' Doc
  -- New: We add a construct representing the union of two documents
  | Doc 'Union' Doc
\end{verbatim}
Relationship of Constructors and Document Operators

The following relationships hold between the constructors and the document operators:

\[
\begin{align*}
\text{Nil} &= \text{nil} \\
\text{s 'Text'} x &= \text{text s} <> x \\
\text{i 'Line'} x &= \text{nest i line} <> x \\
x 'Union' y &= x <|> y
\end{align*}
\]
Example 1(8)

The document

group(
group(
group(  
group(  text "hello" <> line <> text "a")  
<> line <> text "b")  
<> line <> text "c")  
<> line <> text "d")
Example 2(8)

...has the following 5 possible layouts:

```
hello a b c d  hello a b c  hello a b  hello a  hello
   d       c       b       a
```

...
Example 3(8)

**Task:** Print the above document under the constraint that the maximum line width is 5.

⇒ the right-most layout of the previous slide is requested.

**Initial (performance) considerations:**

- Factoring out "hello" of all the layouts in x and y
  \[ "hello" \ 'Text' (" " \ 'Text' x) \ 'Union' (0 \ 'Line' y) \]

- Defining additionally the interplay of (<>️) and nest with Union

\[
(x \ 'Union' y) <>️ z = (x <>️ z) \ 'Union' (y <>️ z)
\]

\[
\text{nest } k \ (x \ 'Union' y) = \text{nest } k \ x \ 'Union' \ \text{nest } k \ y
\]
Example 4(8)

Implementations of `group` and `flatten` can easily be derived:

\[
\begin{align*}
group \Nil &= \Nil \\
group (i \ 'Line' \ x) &= (' ' \ 'Text' \ flatten \ x) \\
&\quad \ 'Union' \ (i \ 'Line' \ x) \\
group (s \ 'Text' \ x) &= s \ 'Text' \ group \ x \\
group (x \ 'Union' \ y) &= group \ x \ 'Union' \ y \\
\end{align*}
\]

\[
\begin{align*}
flatten \Nil &= \Nil \\
flatten (i \ 'Line' \ x) &= (' ' \ 'Text' \ flatten \ x) \\
flatten (s \ 'Text' \ x) &= s \ 'Text' \ flatten \ x \\
flatten (x \ 'Union' \ y) &= flatten \ x \\
\end{align*}
\]
Example 5(8)

Considerations on correctness (similar reasoning as earlier):

Derivation of \( \text{group (i 'Line' x)} \) (see line two) (preserving the invariant required by \text{union})

\[
\begin{align*}
\text{group (i 'Line' x)} & = \{ \text{Definition of Line} \} \\
\text{group (nest i line <> x)} & = \{ \text{Definition of group} \} \\
\text{flatten (nest i line <> x) <|> (nest i line s <> x)} & = \{ \text{Definition of flatten} \} \\
(\text{text " " <> flatten x) <|> (nest i line <> x)} & = \{ \text{Definition of Text, Union, Line} \} \\
(" " 'Text' flatten x) 'Union' (i 'Line' x)
\end{align*}
\]
Correctness considerations (cont’d):

Derivation of \( \text{group (s ‘Text’ x)} \) (see line three)

\[
\text{group (s ‘Text’ x)} \\
= \{ \text{Definition Text } \} \\
\text{group (text s <> x)} \\
= \{ \text{Definition group } \} \\
\text{flatten (text s <> x) <|> (text s <> x)} \\
= \{ \text{Definition flatten } \} \\
(\text{text s <> flatten x) <|> (text s <> x)} \\
= \{ <|> distributes through <|> } \\
\text{text s <> (flatten x <|> x)} \\
= \{ \text{Definition group } \} \\
\text{text s <> group x} \\
= \{ \text{Definition Text } \} \\
\text{s ‘Text’ group x}
\]
Example 7(8)

Selecting the “best” layout:

\[
\begin{align*}
\text{best } w \ k \ \text{Nil} & = \text{Nil} \\
\text{best } w \ k \ (i \ 'Line' \ x) & = i \ 'Line' \ \text{best } w \ i \ x \\
\text{best } w \ k \ (s \ 'Text' \ x) & = s \ 'Text' \ \text{best } w \ (k + \text{length } s) \ x \\
\text{best } w \ k \ (x \ 'Union' \ y) & = \text{better } w \ k \ (\text{best } w \ k \ x) \ (\text{best } w \ k \ y) \\
\text{better } w \ k \ x \ y & = \text{if fits } (w-k) \ x \ \text{then } x \ \text{else } y
\end{align*}
\]

Remark:

- **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.
Example 8(8)

Check, if the first document line stays within the maximum line length \( w \):

\[
\begin{align*}
\text{fits } w \ x \ | \ w < 0 & \quad = \ \text{False} \quad -- \ \text{cannot fit} \\
\text{fits } w \ \text{Nil} & \quad = \ \text{True} \quad -- \ \text{fits trivially} \\
\text{fits } w \ (s \ 'Text' \ x) & \quad = \ \text{fits} \ (w - \text{length} \ s) \ x \\
& \quad \quad \quad -- \ \text{fits if } x \ \text{fits into} \\
& \quad \quad \quad \quad -- \ \text{the remaining space} \\
& \quad \quad \quad \quad -- \ \text{after placing } s \\
\text{fits } w \ (i \ 'Line' \ x) & \quad = \ \text{True} \quad -- \ \text{yes, it fits}
\end{align*}
\]

Last but not least, the output routine (layout remains unchanged):

Select the best layout and convert it to a string:

\[
\text{pretty } w \ x = \text{layout} \ (\text{best } w \ 0 \ x)
\]
Enhancing Performance: A More Efficient Variant

Sources of inefficiency:

1. Concatenation of documents might pile up to the left.
2. Nesting of documents adds a layer of processing to increment the indentation of the inner document.

Problem fix:

- For 1.): Add an explicit representation for concatenation, and generalize each operation to act on a list of concatenated documents.
- For 2.): Add an explicit representation for nesting, and maintain a current indentation that is incremented as nesting operators are processed.
Enhancing Performance: A More Efficient Variant (Cont’d)

Implementing this fix by means of a new implementation of documents:

```haskell
data DOC = NIL                      -- Here is one constructor
  | DOC :<> DOC           -- corresponding to each
  | NEST Int DOC         -- operator that builds a
  | TEXT String          -- document
  | LINE
  | DOC :<>|> DOC
```

Remark:

- In distinction to the previous document type we here use capital letters in order to avoid name clashes with the previous definitions
Implementing the Document Operators

Defining the operators to build a document are straightforward:

\[
\begin{align*}
nil & = \text{NIL} \\
x \ < > \ y & = x :<> y \\
nest \ i \ x & = \text{NEST} \ i \ x \\
text \ s & = \text{TEXT} \ s \\
\text{line} & = \text{LINE}
\end{align*}
\]
Implementing group and flatten

As before, we require the following invariants:

- 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 \( y \).

Definitions of \texttt{group} and \texttt{flatten} are then straightforward:

\[
\begin{align*}
\text{group } x & = \text{flatten } x :<|> x \\
\text{flatten } \text{NIL} & = \text{NIL} \\
\text{flatten } (x :<> y) & = \text{flatten } x :<> \text{flatten } y \\
\text{flatten } (\text{NEST } i \ x) & = \text{NEST } i (\text{flatten } x) \\
\text{flatten } (\text{TEXT } s) & = \text{TEXT } s \\
\text{flatten } \text{LINE} & = \text{TEXT } " " \\
\text{flatten } (x :<|> y) & = \text{flatten } x
\end{align*}
\]
Representation Function

Generating the document from an indentation-afflicted document (“indentation-document pair”):

\[
\text{rep } z = \text{fold } (<>) \text{ nil } [\text{nest } i \ x \ | \ (i,x) \gets z ]
\]
Selecting the “best” Layout

Generalizing the function “best” by composing the old function with the representation function to work on lists of indentation-document pairs:

\[ \text{be } w \ k \ z = \text{best } w \ k \ (\text{rep } z) \quad \text{(Hypothesis)} \]

\[ \text{best } w \ k \ x = \text{be } w \ k \ [(0,x)] \]

where the definition is derived from the old one:

\[ \text{be } w \ k \ [] = \text{Nil} \]
\[ \text{be } w \ k \ ((i,NIL):z) = \text{be } w \ k \ z \]
\[ \text{be } w \ k \ ((i,x :<> y) : z) = \text{be } w \ k \ ((i,x) : (i,y) : z) \]
\[ \text{be } w \ k \ ((i,\text{NEST } j \ x) : z) = \text{be } w \ k \ ((i+j),x) : z) \]
\[ \text{be } w \ k \ ((i,\text{TEXT } s) : z) = s \ 'Text' \ \text{be } w \ (k+\text{length } s) \ z \]
\[ \text{be } w \ k \ ((i,\text{LINE}) : z) = i \ 'Line' \ \text{be } w \ i \ z \]
\[ \text{be } w \ k \ ((i.x :<|> y) : z) = \text{better } w \ k \ (\text{be } w \ k \ ((i.x) : z)) \]
Preparing the XML-Application 1(3)

First some useful supporting functions:

\[
\begin{align*}
x \ <+> \ y & = x \ <+> \ \text{text} " " \ <+> \ y \\
x \ <+> \ y & = x \ <+> \ \text{line} \ <+> \ y \\
\end{align*}
\]

\[
\begin{align*}
\text{folddoc} \ f \ [\_] & = \text{nil} \\
\text{folddoc} \ f \ [x] & = x \\
\text{folddoc} \ f \ (x:xs) & = f \ x \ (\text{folddoc} \ f \ xs) \\
\end{align*}
\]

\[
\begin{align*}
\text{spread} & = \text{folddoc} \ (\ <+>) \\
\text{stack} & = \text{folddoc} \ (\ <+>)
\end{align*}
\]
Preparing the XML-Application 2(3)

Further supportive functions:

-- An often recurring output pattern
bracket l x r = group (text l <>
    nest 2 (line <> x) <>
    line <> text r)

-- Abbreviation of the alternative tree
-- layout function
showBracket’ ts = bracket "[" (showTrees’ ts) "]"

-- Filling up lines (using words out of the
-- Haskell Standard Lib.)
x <+/> y = x <> (text " " :<|> line) <> y
fillwords = folddoc ( <+/> ). map text . words
fill, a variant of fillwords
    \(\rightsquigarrow\) collapses a list of documents to a single document.

\[
\begin{align*}
\text{fill } [] & = \text{nil} \\
\text{fill } [x] & = x \\
\text{fill } (x:y:zs) & = (\text{flatten } x \oplus \text{fill } (\text{flatten } y : zs)) :<|> \\
& \quad (x \not<>/ \text{fill } (y : zs))
\end{align*}
\]
Application

Printing XML-documents (simplified syntax):

```haskell
data XML = Elt String [Att] [XML] | Txt String

data Att = Att String String

showXML x = folddoc (<>)(showXMLs x)

showXMLs (Elt n a [])
    = [text "<" <> showTag n a <> text "/>"

showXMLs (Elt n a c)
    = [text "<" <> showTag n a <> text ">
        showFill showXMLs c <>
        text "/" <> text n <> text ">
    ]

showXMLs (Txt s) = map text (words s)

showAtts (Att n v)
    = [text n <> text "+=" <> text (quoted v)]
```
Application (Cont’d)

Continuation:

```haskell
quoted s = "\"" ++ s ++ "\""

showTag n a = text n <> showFill showAtts a

showFill f [] = nil
showFill f xs
  = bracket "" (fill (concat (map f xs))) ""
```
1st XML Example

...for a given maximum line length of 30 letters:

```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>
```
2nd XML Example

...for a given maximum line length of 60 letters:

<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>
3rd XML Example

...after dropping of flatten in fill:

```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>
```

...start and close tags are crammed together with other text
⇝ less beautifully than before.
Summing up: Why “prettier” than “pretty”? 

The below pretty printer library proposed by John Hughes is widely recognized as a standard:


From a technical perspective, the library of John Hughes enjoys the following characteristics:

- There are two ways (horizontal and vertical) to concatenate documents, one of which
  - without unit (vertical)
  - with right-unit but no left-unit (horizontal)
Summing up (Cont’d)

Philip Wadler considers his “Prettier Printer” an improvement of John Hughes’ pretty printer library.

From a technical perspective, a distinguishing feature of the “Prettier Printer” proposed by Philip Wadler is:

- There is only a single way to concatenate documents that is
  - associative
  - with a left-unit and a right-unit.

Moreover, John Hughes’ pretty printer library

- consists of ca. 40% more code,
- is ca. 40% slower

as the “prettier printer” of Philip Wadler’s proposal.
Summary of the Code 1(12)


```
infixr 5:<|>
infixr 6:<>
infixr 6 <>

data DOC = NIL
  |  DOC :<> DOC
  |  NEST Int DOC
  |  TEXT String
  |  LINE
  |  DOC :<|> DOC

data Doc = Nil
  |  String ’Text’ Doc
  |  Int ’Line’ Doc
```
Summary of the Code 2(12)

\[
\begin{align*}
nil &= \text{NIL} \\
x <> y &= x :<> y \\
nest i x &= \text{NEST } i \ x \\
text s &= \text{TEXT } s \\
line &= \text{LINE} \\
group x &= \text{flatten } x :<|> x \\
\text{flatten } \text{NIL} &= \text{NIL} \\
\text{flatten } (x :<> y) &= \text{flatten } x:<> \text{flatten } y \\
\text{flatten } (\text{NEST } i \ x) &= \text{NEST } i \ (\text{flatten } x) \\
\text{flatten } (\text{TEXT } s) &= \text{TEXT } s \\
\text{flatten } \text{LINE} &= \text{TEXT } " " \\
\text{flatten } (x :<|> y) &= \text{flatten } x
\end{align*}
\]
Summary of the Code 3(12)

```haskell
layout Nil = ""
layout (s 'Text' x) = s ++ layout x
layout (i 'Line' x) = '\n': copy i ' ' ++ layout x

copy i x = [x | _ <- [1..i]]
```
Summary of the Code 4(12)

```
best w k x       = be w k [(0,x)]
be w k []        = Nil
be w k ((i,NIL):z) = be w k z
be w k ((i,x :<> y) : z)
    = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)
    = s 'Text' be w (k+length s) z
be w k ((i,LINE) : z)    = i 'Line' be w i z
be w k ((i,x :<|> y) : z)
    = better w k (be w k ((i.x) : z))
        (be w k (i,y) : z))

better w k x y
    = if fits (w-k) x then x else y
```
Summary of the Code 5(12)

fits w x | w<0 = False
fits w Nil = True
fits w (s 'Text' x) = fits (w - length s) x
fits w (i 'Line' x) = True

pretty w x = layout (best w 0 x)

-- Utility functions
x <+> y = x <> text " " <> y
x </> y = x <> line <> y

folddoc f [] = nil
folddoc f [x] = x
folddoc f (x:xs) = f x (folddoc f xs)
Summary of the Code 6(12)

\[
\begin{align*}
\text{spread} &= \text{folddoc} (\langle+\rangle) \\
\text{stack} &= \text{folddoc} (\langle/\rangle)
\end{align*}
\]

\[
\text{bracket} l \ x \ r = \text{group} \ (\text{text} \ l \ \langle\rangle \\
\quad \langle\text{nest} \ 2 \ (\text{line} \ \langle\rangle \ x) \ \langle\rangle \\
\quad \langle\text{line} \ \langle\rangle \ \text{text} \ r) \\
x \ \langle+/\rangle \ y &= x \ \langle\rangle \ (\text{text} \ " \ " \ \langle|\rangle \ \text{line}) \ \langle\rangle \ y
\]

\[
\begin{align*}
\text{fillwords} &= \text{folddoc} (\langle+/\rangle) \ . \ \text{map} \ \text{text} \ . \ \text{words} \\
\text{fill} \ [\] &= \text{nil} \\
\text{fill} \ [x] &= x \\
\text{fill} \ (x:y:zs) &= (\text{flatten} \ x \ \langle+/\rangle \ \text{fill} \ (\text{flatten} \ y : \ zs)) \\
&\quad \langle|\rangle \ (x \ \langle/\rangle \ \text{fill} \ (y : \ zs)
\end{align*}
\]
Summary of the Code 7(12)

-- Tree example

data Tree = Node String [Tree]

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
Summary of the Code 8(12)

```haskell
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
```
Summary of the Code 9(12)

```haskell
tree = Node "aaa" [ Node "bbbb" [ Node "ccc" [], Node "dd" []
                          ],
                 Node "eee" [],
                 Node "ffff" [ Node "gg" [], Node "hhh" [], Node "ii" []
                               ]]

testtree w = putStrLn (pretty w (showTree tree))
testtree' w = putStrLn (pretty w (showTree' tree))
```
Summary of the Code 10(12)

-- XML Example

```haskell
data XML = Elt String [Att] [XML] | Txt String

data Att = Att String String

showXML x = folddoc (<>)(showXMLs x)

showXMLs (Elt n a [])
    = [text "<" <> showTag n a <> text "/>"

showXMLs (Elt n a c)
    = [text "<" <> showTag n a <> text ">" <>
        showFill showXMLs c <>
        text "/" <> text n <> text ">"]

showXMLs (Txt s) = map text (words s)
```
Summary of the Code 11(12)

```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))) ""
```
Summary of the Code 12(12)

xml =
    Elt "p" [Att "color" "red",
               Att "font" "Times",
               Att "size" "10"
           ] [ Txt "Here is some",
                Elt "em" [] [ Txt "emphasized"],
                Txt "text.",
                Txt "Here is a",
                Elt "a" [ Att "href" "http://www.eg.com/"
                         [ Txt "link" ],
                         Txt "elsewhere." ]

    ]

testXML w = putStrLn (pretty w (showXML xml))
On an early imperative “Pretty Printer:”


...and a functional realization of it:

Overview on the evolution of a **Pretty Printer Library** and origin of the development of the **Prettier Printers** proposed by Philip Wadler:


...a variant is implemented in the Glasgow Haskell Compiler:

Chapter 15: Further Reading (1)


Chapter 15: Further Reading (2)


Chapter 15: Further Reading (3)


Chapter 16
Functional Reactive Programming
Motivation

Hybrid systems are systems that are composed of

- continuous and
- discrete

components.
Mobile Robots

Mobile robots are special hybrid systems:

- **From a physical perspective:**
  - **Continuous components**: Voltage-controlled motors, batteries, range finders,...
  - **Discrete components**: Microprocessors, bumper switches, digital communication,...

- **From a logical perspective:**
  - **Continuous notions**: Wheel speed, orientation, distance from a wall,...
  - **Discrete notions**: Running into another object, receiving a message, achieving a goal,...
Objective of this Chapter

Designing and implementing two

▶ imperative-style languages for controlling robots which will be done in terms of a simulation (in order to allow running the simulations at home without having to buy (possibly expensive) robots first).

This will deliver two examples of a

▶ domain specific language (DSL).

Simultaneously, it yields a nice application of the

▶ higher-order type (constructor) classes
  ▶ Functor
  ▶ Monad
  ▶ Arrows
Reading for this Chapter

For Chapter 16.1:


  ⇝ using monads

For Chapter 16.2:


  ⇝ using arrows

Note: Chapter 16.1 and 16.2 are independent of each other; they do not build on each other.
Chapter 16.1
An Imperative Robot Language
Our robots’ world:

...is two-dimensional with gold coins as treasures!
The World of Robots – Illustration (2)

In more detail:

The **world** the robots live in
  - is a finite **two-dimensional grid** of square form
    - equipped with **walls**
    - that might form **rooms** and might have **doors**
    - with placed **gold coins** on some grid points

The preceding illustration shows an example of a
  - robot’s world with one room full of **gold coins**: **Eldorado**!
  - and a robot sitting in the centre of the world ready for exploring it!
The World of Robots – Illustration (3)

A robot’s job:

...exploring the world, collecting treasures, leaving footprints!
The World of Robots – Illustration (4)

In more detail:

A robot’s job

- is to explore its world, to collect the treasures in it, and to leave footprints of its exploration, i.e.,
  - to systematically stroll through its world, e.g., in the form of an outward-oriented spiral
  - picking up the gold coins it finds and saving them in its pocket
  - dropping gold coins at some grid points
  - marking its way with a colored pen
Objective

...enabling the robots to explore and shape their world!

In other words, we would like to write programs such as:

(1) drawSquare =
    do penDown
    move
    turnRight
    move
    turnRight
    move

(2) moveToWall =
    while (isnt blocked)
        do move

(3) getRich =
    while (isnt blocked)
        $ do move
        checkAndPickCoin
Modeling the World

Modeling the world our robots live and act in:

```haskell
type Grid = Array Position [Direction]

type Position = (Int,Int)

data Direction = North | East | South | West
    deriving (Eq, Show, Enum)
```
Modeling the Robots (1)

The internal states of the robots are made up by:

1. Robot position
2. Robot orientation
3. Pen status (up or down)
4. Pen color
5. Placement of gold coins on the grid
6. Number of coins in the robot’s pocket
Modeling the Robots (2)

Modeling the internal states of the robots:

```haskell
data RobotState = RobotState
  { position :: Position
  , facing :: Direction
  , pen :: Bool
  , color :: Color
  , treasure :: [Position]
  , pocket :: Int
  }

deriving Show

where

data Color = Black | Blue | Green | Cyan
            | Red | Magenta | Yellow | White

deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```
Remarks (1)

Note that the above definition takes advantage of Haskell’s field-label syntax:

- Field labels (here position, facing, pen, color, treasure, pocket) allow access to components by names instead of position without necessitating specific selector functions.
Remarks (2)

Robot states could have been equivalently be defined without referring to field label syntax:

```haskell
data RobotState = RobotState
    Position
    Direction
    Bool
    Color GM
    [Position]
    Int
    deriving Show
```

...losing the advantage of accessing fields by names.
Remarks (3)

Illustrating the usage of field labels: Generating, accessing, modifying values of state components.

Example 1: Generating field values

The definition

\[
s_1 = \text{RobotState} \ (0,0) \ \text{East} \ \text{True} \ \text{Green} \\
[\ (2,3), (7,9), (12,42)\ ] \ 2 :: \text{RobotState}
\]

is equivalent to

\[
s_2 = \text{RobotState} \ {\{ \ 	ext{position} = (0,0) \\
, \ \text{facing} = \text{East} \\
, \ \text{pen} = \text{True} \\
, \ \text{color} = \text{Green} \\
, \ \text{treasure} = [\ (2,3), (7,9), (12,42)\ ] \\
, \ \text{pocket} = 2 \\
\} :: \text{RobotState}
\]
Remarks (4)

Advantages of using field label syntax:

▶ It is more “informative.”
▶ The order of fields gets irrelevant.

For example: The definition of $s_3$

$$s_3 = \text{RobotState}$$

$$\{ \text{position} = (0,0)$$

$$, \text{pocket} = 2$$

$$, \text{pen} = \text{True}$$

$$, \text{color} = \text{Green}$$

$$, \text{treasure} = [(2,3),(7,9),(12,42)]$$

$$, \text{facing} = \text{East}$$

$$\} :: \text{RobotState}$$

is equivalent to that of $s_2$. 
Remarks (5)

Example 2: Accessing field values

position s2  --->>  (0,0)
treasure s3  --->>  [(2,3),(7,9),(12,42)]
color s3  --->>  Green

Example 3: Modifying field values

s3 { position = (22,43), pen = False }
   --->>  RobotState { position = (22,43)
                        , facing = East
                        , pen = False
                        , color = Green
                        , treasure = [(2,3),(7,9),(12,42)]
                        , pocket = 2
                  } :: RobotState
Example 4: Using field names in patterns

\[ \text{jump (RobotState \{ position = (x,y) \}) = (x+2,y+1)} \]
Robots as a Member of Type Class Monad

Defining **Robot** as an algebraic data type

```haskell
newtype Robot a
  = Robot (RobotState -> Grid
               -> Window -> IO (RobotState,a))
```

...allows making **Robot** an instance of type class **Monad**:

```haskell
instance Monad Robot where
  Robot sf0 >>= f = Robot $ \s0 g w -> do
      (s1,a1) <- sf0 s0 g w
      let Robot sf1 = f a1
      (s2,a2) <- sf1 s1 g w
      return (s2,a2)
  return a = Robot (\s _ _ -> return(s,a))
```
Remarks (1)

Note that

```haskell
instance Monad Robot where
  return a = Rob (\s _ _ -> return(s,a))
  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)
```

requires function application “$”, not function composition “.”
(For clarity, Robot has been replaced by Rob (cp. next slide)).
Remarks (2)

The `Window` argument

```haskell
newtype Robot a = Rob (RobotState -> Grid -> Window -> IO (RobotState,a))
```

...allows to specify the `window`, in which the graphics is displayed.
Robots – Simulation and Control

The implementation environment:

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.
Key insight:

- Taking state as input
- Possibly querying the state in some way
- Returning a possibly modified state

...makes the imperative nature of IRL commands obvious.
IRL – The Imperative Robot Language (2)

IRL commands and their implementation:

- Commands not related to graphics:

  `right, left :: Direction -> Direction`

  `right d = toEnum (succ (mod (fromEnum d) 4))`

  `left d = toEnum (pred (mod (fromEnum d) 4))`

- Supporting functions for updating and querying states:

  `updateState :: (RobotState -> RobotState) -> Robot ()`

  `updateState u = Robot (\s _ _ -> return (u s, ()))`

  `queryState :: (RobotState -> a) -> Robot a`

  `queryState q = Robot (\s _ _ -> return (s, q s))`
The Type Class Enum (1)

...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]

suc = 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]

...implementation is type-dependent
The Type Class Enum (2)

The following equivalences hold:

\[
\begin{align*}
\text{enumFrom } n & \quad \sim \quad [n..] \\
\text{enumFromThen } n \ n' & \quad \sim \quad [n,n'..] \\
\text{enumFromTo } n \ m & \quad \sim \quad [n..m] \\
\text{enumFromThenTo } n \ n' \ m & \quad \sim \quad [n,n'..m]
\end{align*}
\]

Example:

\[
\begin{align*}
data \ \text{Color} & = \ \text{Red} \ | \ \text{Orange} \ | \ \text{Yellow} \ | \ \text{Green} \\
& \quad | \ \text{Blue} \ | \ \text{Indigo} \ | \ \text{Violet}
\end{align*}
\]

\[
\begin{align*}
\text{instance \ Enum \ Color \ where} \\
& \quad \ldots
\end{align*}
\]

\[
\begin{align*}
[\text{Red..Green}] & \quad \rightarrow \ [\text{Red, Orange, Yellow, Green}] \\
[\text{Red, Yellow..}] & \quad \rightarrow \ [\text{Red, Yellow, Blue, Violet}] \\
\text{fromEnum Blue} & \quad \rightarrow \ 4 \\
\text{toEnum 3} & \quad \rightarrow \ \text{Green}
\end{align*}
\]
Commands for robot orientation:

```haskell
turnLeft :: Robot ()
turnLeft =
    updateState (\s -> s {facing = left (facing s)})

turnRight :: Robot ()
turnRight =
    updateState (\s -> s {facing = right (facing s)})

turnTo :: Direction -> Robot ()
turnTo d = updateState (\s -> s {facing = d})

direction :: Robot Direction
direction = queryState facing
```
Commands for blockade checking:

```haskell
blocked :: Robot Bool
blocked =
    Robot $ \s g _ ->
        return(s, facing s 'notElem' (g 'at' position s))
```
Commands for moving a robot:

move :: Robot ()
move =
cond1 (isnt blocked)
(Rob $ \ s \ w \ -> \ do
let newPos = movePos (position s) (facing s)
graphicsMove w s newPos
return (s {position = newPos}, ())
)

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)
Commands for using the pen:

```haskell
penUp :: Robot ()
penUp = updateState (\s -> s {pen = False})

penDown :: Robot ()
penDown = updateState (\s -> s {pen = True})

setPenColor :: Color -> Robot ()
setPenColor c = updateState (\s -> s {color = c})
```
Commands for handling coins:

```haskell
onCoin :: Robot Bool
onCoin = queryState \( \text{\textbackslash s \rightarrow} \)

    position s 'elem' treasure s

coins :: Robot Int
coins = queryState pocket
```
More commands for handling coins:

```haskell
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 – The Imperative Robot Language (8)
More commands for handling coins:

```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}, ())
    )
```
Logic and Control (1)

Logic and control functions:

```haskell
cond :: Robot Bool -> Robot a -> Robot a
cond p c a = do pred <- p
    if pred then c else a

cond1 p c = cond p c (return ())

while :: Robot Bool -> Robot () -> Robot ()
while p b = cond1 p (b >> while p b)

(||*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 ||* b2 = do p <- b1
    if p then return True
    else b2
```
Logic and Control (2)

▶ Logic and control functions (cont’d):

\[
\text{(&&*)} :: \text{Robot Bool} \rightarrow \text{Robot Bool} \rightarrow \text{Robot Bool} \\
\text{b1 \&\&* b2} = \text{do p <- b1} \\
\text{\hspace{1cm} if p then b2} \\
\text{\hspace{1cm} else return False}
\]

\[
isnt :: \text{Robot Bool} \rightarrow \text{Robot Bool} \\
isnt = \text{liftM not}
\]

\[
(>*) :: \text{Robot Int} \rightarrow \text{Robot Int} \rightarrow \text{Robot Bool} \\
(>*) = \text{liftM2 (>)}
\]

\[
(<*) :: \text{Robot Int} \rightarrow \text{Robot Int} \rightarrow \text{Robot Bool} \\
(<*) = \text{liftM2 (<)}
\]
The higher-order functions \texttt{liftM} and \texttt{liftM2} are defined in the library \texttt{Monad} (as well as \texttt{liftM3},...,,\texttt{liftM5}): 

\[
\texttt{liftM} \quad :: \quad \text{(Monad m) \Rightarrow (a \rightarrow b) \rightarrow (m a \rightarrow m b)}
\]

\[
\text{liftM } f = \ \lambda a \rightarrow \ do \ a' \leftarrow a \\
\hspace{1cm} \text{return } (f \ a')
\]

\[
\texttt{liftM2} \quad :: \quad \text{(Monad m) \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow (m a \rightarrow m b \rightarrow m c)}
\]

\[
\text{liftM2 } f = \ \lambda a \ b \rightarrow \ do \ a' \leftarrow a \\
\hspace{1cm} b' \leftarrow b \\
\hspace{1cm} \text{return } (f \ a' \ b')
\]
Logic and Control (4)

Note:

- Basing the implementations of `isnt`, `(>*)` and `(<<*)` on `liftM` and `liftM2` allows to dispense the usage of special `lift` functions.
- No basing of the implementations of `(||*)` and `(&&*)` on `liftM2` in order to avoid (unnecessary) strictness in their second arguments.
Further Data Structures

colors :: Array Int Color

colors = array (0,7)
    [(0,Black),(1,Blue),(2,Green),(3,Cyan),
     (4,Red),(5,Magenta),(6,Yellow),(7,White)]

where (as a reminder!)

data Color = Black | Blue | Green | Cyan
            | Red  | Magenta | Yellow | White

deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)

Note:

▶ Color is defined as in the library Graphics.
▶ Equivalently we could have defined more concisely:

colors :: Array Int Color

    colors = array (0,7) (zip [0..7] [Black..White])
Shaping the Robots’ Initial World \(g_0\)

The robots’ world is a grid of type `Array`:

\[
\text{type Grid = Array Position [Direction]}
\]

We can access the grid points using:

\[
\text{at :: Grid -> Position -> [Direction]}
\]
\[
\text{at = (!)}
\]

The size of the initial grid \(g_0\) is given by:

\[
\text{size :: Int}
\]
\[
\text{size = 20}
\]

with

- centre \((0,0)\) and
- corners \((size, size), ((-size), size), ((-size), (-size))\ and \((size, (-size))\).
The Initial World $g_0$ (1)

...and the 4 surrounding walls (no walls inside):

- **Inner points of $g_0$** are given by:
  \[
  \text{interior} = [\text{North}, \text{South}, \text{East}, \text{West}]
  \]

- **Extremal points** on the grid borders (north border, north-east corner, etc.) are given by:
  \[
  \begin{align*}
  \text{nb} & = [\text{South}, \text{East}, \text{West}] \quad -- \text{nb: north border} \\
  \text{sb} & = [\text{North}, \text{East}, \text{West}] \\
  \text{eb} & = [\text{North}, \text{South}, \text{West}] \\
  \text{wb} & = [\text{North}, \text{South}, \text{East}] \quad -- \text{wb: west border} \\
  \text{nwc} & = [\text{South}, \text{East}] \quad -- \text{nwc: northwest corner} \\
  \text{nec} & = [\text{South}, \text{West}] \\
  \text{swc} & = [\text{North}, \text{East}] \\
  \text{sec} & = [\text{North}, \text{West}] \quad -- \text{sec: southeast corner}
  \end{align*}
  \]
The Initial World $g_0$ (2)

This allows: ...enumerating inner and border grid points using a list comprehension:

$$g_0 :: \text{Grid}$$
$$g_0 = \text{array} ((-\text{size}, -\text{size}), (\text{size}, \text{size}))$$
$$(([((i, \text{size}), \text{nb}) | i \leftarrow r ] ++$$
$$([((i, -\text{size}), \text{sb}) | i \leftarrow r ] ++$$
$$([((\text{size}, i), \text{eb}) | i \leftarrow r ] ++$$
$$([((-\text{size}, i), \text{wb}) | i \leftarrow r ] ++$$
$$([((\text{size}, i), \text{eb}) | i \leftarrow r ] ++$$
$$([((i,j), \text{interior}) | i \leftarrow r, j \leftarrow r ] ++$$
$$([((\text{size}, \text{size}), \text{nec}), ((\text{size}, -\text{size}), \text{sec}),$$
$$((-\text{size}, \text{size}), \text{nwc}),$$
$$((-\text{size}, -\text{size}), \text{swc}])$$
where $r = [1-\text{size}..\text{size}-1]$
The new World $g_1$ that extends World $g_0$ (1)

...evolves from building new walls using the array library functions (\(/\)):

\[
(\/) :: \text{Ix } a \Rightarrow \text{Array } a \ b \rightarrow [(a,b)] \rightarrow \text{Array } a \ b
\]

Example: Application of (\(/\))

- Reversing the positions of “black” and “white” in colors:

\[
\text{colors} \ ,(\/) \ [(0,\text{White}),(7,\text{Black})] \\
\rightarrow \text{array } (0,7) \\
\quad [(0,\text{White}),(1,\text{Blue}),(2,\text{Green}),(3,\text{Cyan}), \\
\quad (4,\text{Red}),(5,\text{Magenta}),(6,\text{Yellow}), \\
\quad (7,\text{Black})] :: \text{Array } \text{Integer } \text{Color}
\]
The new World $g_1$ that extends World $g_0$ (2)

Supporting functions for building new walls:

-- Building horizontal and vertical walls
mkHorWall, mkVerWall ::
  Int -> Int -> Int -> [(Position, [Direction])]

-- Building west/east-oriented walls
-- leading from (x1,y) to (x2,y)
mkHorWall x1 x2 y
  = [((x,y), nb) | x <- [x1..x2]] ++
    [((x,y+1), sb) | x <- [x1..x2]]

-- Building north/south-oriented walls
-- leading from (x,y1) to (x,y2)
mkVerWall y1 y2 x
  = [((x,y), eb) | y <- [y1..y2]] ++
    [((x+1,y), wb) | y <- [y1..y2]]
The new World $g_1$ that extends World $g_0$ (3)

World $g_1$ evolves from world $g_0$ by

- building a west/east-oriented wall leading from $(-5, 10)$ to $(5, 10)$:

$$g_1 :: \text{Grid}$$

$$g_1 = g_0//\text{mkHorWall} (-5) 5 10$$
The World $g_2$ that extends $g_0$ (1)

Supporting functions for building a “room:”

$$mkBox :: \text{Position} \rightarrow \text{Position}$$
$$\rightarrow [(\text{Position}, [\text{Direction}])]$$

$$mkBox (x_1, y_1) (x_2, y_2)$$
$$= mkHorWall (x_1+1) x_2 y_1 ++$$
$$mkHorWall (x_1+1) x_2 y_2 ++$$
$$mkVerWall (y_1+1) y_2 x_1 ++$$
$$mkVerWall (y_1+1) y_2 x_2$$

Note:

- The above function creates two field entries for each of the four inner corners.
- After creation the value of these entries are still undefined.
- Using the function `accum` allows initializing these entries on-the-fly of their creation:

$$\text{accum} :: (Ix a) => (b \rightarrow c \rightarrow b)$$
$$\rightarrow \text{Array a b} \rightarrow [(a,c)] \rightarrow \text{Array a b}$$
The World g2 that extends g0 (2)

Recall the function \texttt{accum}:

\[
\texttt{accum :: (Ix a) => (b -> c -> b) \rightarrow Array a b \rightarrow [(a,c)] \rightarrow Array a b}
\]

The function \texttt{accum}

\begin{itemize}
\item is quite similar to the function \texttt{(//)}.
\item in case of replicated entries the function of the first argument is used for resolving conflicts.
\item the List-library function \texttt{intersect} is suitable for this for the case of our example:
\end{itemize}

Example:

\[
\text{[South, East, West] 'intersect'} \\
\quad [\text{North, South, West}] \rightarrow [\text{South, West}]
\]

which corresponds to a northeast corner.
The World $g_2$ that extends $g_0$ (3)

Example: Building a room with $(-10,5)$ as lower left corner and $(-5,10)$ as upper right corner

- using \texttt{accum und intersect}.

World $g_2$ then extends world $g_0$:

\[
g_2 :: \text{Grid} \\
g_2 = \text{accum intersect} \ g_0 \ (\text{mkBox} \ (-15,8) \ (2,17))
\]
The World $g_3$ that extends $g_2$

Continuing the example: Adding a door (to the middle of the top wall of the room)

- using \texttt{accum und union}.

World $g_2$ evolves to world $g_3$:

\[
g_3 ::= \text{Grid} \\
g_3 = \text{accum union } g_2 \left[(-7,17), \text{ interior}\right], \left[(-7,18), \text{ interior}\right]
\]
Animation: Robot Graphics (1)

Animation

- by means of incrementally updating the world.

To this end we make use of the function:

\[
\text{drawLine} :: \text{Window} \rightarrow \text{Color} \rightarrow \text{Point} \rightarrow \text{Point} \rightarrow \text{IO} ()
\]

\[
\text{drawLine} \ w \ c \ p1 \ p2 = \text{drawInWindowNow} \ w \ (\text{withColor} \ c \ (\text{line} \ p1 \ p2))
\]

which makes use of the Graphics-library function \text{drawInWindowNow}.
Animation: Robot Graphics (2)

The incremental update of the world must ensure

- absence of interferences of graphics actions.

To this end we assume:

1. Grid points are 10 pixels apart.
2. Wall are drawn halfway between grid points.
3. Lines drawn by a robot’s pen directly connects two grid points.
4. Coins are drawn as yellow circles just to the above and to the left of a grid point.
5. Erasing coins is done by drawing black circles over already existing yellow ones.
Using the below top level constants ensures the absence of interferences:

\[
\begin{align*}
\text{d} & \quad :: \ Int \\
\text{d} & \quad = \ 5 \quad \quad \quad \quad \quad \text{-- half the distance} \\
& \quad \quad \quad \quad \quad \text{-- between grid points}
\end{align*}
\]

\[
\begin{align*}
\text{wc, cc} & \quad :: \ Color \\
\text{wc} & \quad = \ \text{Blue} \quad \quad \quad \quad \text{-- color of walls} \\
\text{cc} & \quad = \ \text{Yellow} \quad \quad \quad \quad \text{-- color of coins}
\end{align*}
\]

\[
\begin{align*}
\text{xWin, yWin} & \quad :: \ Int \\
\text{xWin} & \quad = \ 600 \\
\text{yWin} & \quad = \ 500
\end{align*}
\]
Animation in Action (1)

Putting it all together.

User-control of program progress by the program’s awaiting the user’s hitting the spacebar:

```haskell
spaceWait :: Window -> IO ()
spaceWait w = do k <- getKey w
    if k==' '
    then return ()
    else spaceWait w
```
Animation in Action (2)

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
```
Animation in Action (3)

Intuitively, `runRobot` causes:

- Opening a window
- Drawing a grid
- Drawing the coins
- Waiting for the user to hit the spacebar
- Continuing running the program with starting state $s$ and grid $g$
Animation in Action (4)

Fixing a suitable starting state:

```haskell
s0 :: RobotState
s0 = RobotState {position = (0,0)
                 , pen = False
                 , color = Red
                 , facing = North
                 , treasure = tr
                 , pocket = 0
            }

tr :: [Position]
tr = [(x,y) | x <- [-13,-11..1], y <- [9,11..15]]

...i.e., all coins are placed inside of the room of grid g3.
```
Animation in Action (5)

Last but not least:

main = runRobot spiral s0 g0

...leads to the “spiral” example shown and discussed at the beginning of this chapter:
Additional Supporting Functions (1)

For drawing a grid:

```haskell
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]]
```

Additional Supporting Functions (2)

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 ()
Additional Supporting Functions (3)

For dropping and erasing 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)))

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)))
Further Supporting Functions (4)

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 ()
      getWinowTick w
Further Supporting Functions (5)

trans :: Position -> Point
trans (x,y) = (div xWin 2+2*d*x, div yWin 2-2*d*y)

getWindowTick :: Window -> IO ()
-- causes a short delay after each robot move

bounds :: Ix a => Array a b -> (a,a)
-- from the Array-library; yields the bounds
-- of an array argument
Chapter 16.2
Robots on Wheels
Outline

In this chapter, we consider a simulation of

- mobile robots (called Simbots) by means of functional reactive programming.

The simulation will make use of

- the type class Arrow that is another example of a type constructor class generalizing the concept of a monad.
Setting the Scene (1)

Mobile robots are assumed to be configured 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.
Setting the Scene (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.”
Simulation Code for “Robots on Wheels”

...can be down-loaded at the Yampa homepage at

www.haskell.org/yampa

In the following we will consider some code snippets.
Preliminaries

- **Simbot** is short for *simulated robot*.
- **SF** denotes the type *signal function*. It is defined in Yampa, which also provides a number of primitive signal functions together with a set of special composition operators (or “combinators”) allowing the construction of more complex signal functions (abstract data type).
- **SF** is an instance of the type constructor class *Arrow*.
- Signal functions, i.e., values of type **SF**, are *signal transformers*, i.e., functions that map signals to signals.
- Signals are not allowed as first-class values in Yampa. Signals can only be manipulated by means of signal functions to avoid time- and space-leaks.
Robot Controller

type Time = Double

type Signal a~ = Time -> a

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

    type RobotType = String
    type RobotId = Int
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
  | OTSimbotB otRId :: RobotId, otPos :: Position2, otHdng :: Heading
Structure of the Program

module MyRobotShow where
  import AFrob
  import AFrobRobotSim

main :: IO ()
main = runSim (Just world) rcA rcB

world :: WorldTemplate
world = ...

rcA :: SimbotController -- controller for simbot A’s
rcA = ...

rcB :: SimbotController -- controller for simbot B’s
rcB = ...
Robot Simulation in Action

Running the robot simulation:

```haskell
runSim :: Maybe WorldTemplate
       -> SimbotController
       -> SimbotController -> IO ()
```
Robot Control

rcA :: SimbotController
rcA rProps =
    case rrpId rProps of
        1 -> rcA1 rProps
        2 -> rcA2 rProps
        3 -> rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
Robot Actions: Control Programs (1)

A stationary robot:

\[
\text{rcStop} :: \text{SimbotController} \\
\text{rcStop}_r = \text{constant} \ (\text{mrFinalize} \ \text{ddBrake})
\]

A blind robot moving at constant speed:

\[
\text{rcBlind1}_r = \\
\quad \text{constant} \ (\text{mrFinalize} \ \text{ddVelDiff} \ 10 \ 10)
\]

A blind robot moving at half the maximum speed:

\[
\text{rcBlind2} \ rps = \\
\quad \text{let} \ \text{max} = \text{rpWSMax} \ rps \\
\quad \text{in} \ \text{constant} \ (\text{mrFinalize} \ \text{ddVelDiff} \ (\text{max}/2) \ (\text{max}/2))
\]
Robot Actions: Control Programs (2)

A robot rotating at a pre-given speed:

\[
\text{rcTurn} :: \text{Velocity} \to \text{SimbotController}
\]

\[
\text{rcTurn \ vel \ rps =}
\]

\[
\text{let \ vMax = rpWSMax \ rps}
\]

\[
\text{rMax = 2 * (vMax - vel) / rpDiameter \ rps}
\]

\[
\text{in \ constant \ (mrFinalize \$ \ ddVelTR \ vel \ rMax)}
\]
Classes of Robots (1)

- Usually, there are different types of robots with different features (2 wheels, 3 wheels, camera, sonar, speaker, blinker, etc.)
- The kind of a robot is fixed by its input and output types.

The kind of robots is encoded in input and output classes together with the functions operating on them.
Kinds of Robots (2)

Input classes and functions operating on them:

class HasRobotStatus i where
  rsBattStat :: i -> BatteryStatus -- Current battery
                -- status
  rsIsStuck :: i -> Bool  -- Currently stuck
                -- or not stuck

data BatteryStatus = BSHigh | BSLow | BSCritical
  deriving (Eq, Show)

-- 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 ()))
Classes of Robots (3)

class HasOdometry where
   odometryPosition :: i -> Position2 -- Current
                   -- position
   odometryHeading :: i -> Heading -- Current
                      -- 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
Classes of Robots (4)

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)
Classes of Robots (5)

**Output classes** and functions operating on them:

```haskell
class MergeableRecord o => HasDiffDrive o where
  ddBrake :: MR o -- Brake both wheels
  ddVelDiff :: Velocity -> Velocity
                     -> MR o -- Set wheel
                      -- velocities

  ddVelTR :: Velocity -> RotVel
             -> MR o -- Set veloc.
                         -- and rotat.

class MergeableRecord o => HasTextConsoleOutput o where
  tcoPrintMessage :: Event String -> MR o
```

1189/1368
Arrows and Mobile Robots

SF is an instance of class \textit{Arrow}:

\[
SF \ a \ b \ = \ \text{Signal} \ a \ \to \ \text{Signal} \ b
\]

\[
\text{Signal} \ a \ = \ \text{Time} \ \to \ a
\]

\text{type} \ \text{Time} \ = \ \text{Double}

Recall:

\begin{itemize}
  \item Values of type SF are \textit{signal transformers} resp. \textit{signal functions}; therefore the name SF.
\end{itemize}
Chapter 16.3
More on the Background of FRP
Origins of FRP

The origins of functional reactive programming (FRP) lie in functional reactive animation (FRAn):


Seminal Works on FRP

Seminal works on function reactive programming (FRP):


Applications of FRP (1)

On Functional Animation Languages (FAL):


On Functional Reactive Robotics (FRob):


Applications of FRP (2)

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: Further Reading (1)


Chapter 16: Further Reading (2)


Chapter 16: Further Reading (3)


Chapter 16: Further Reading (4)


Chapter 16: Further Reading (5)


Chapter 16: Further Reading (6)


Chapter 16: Further Reading (7)


Part VI

Extensions and Prospectives
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

- Kapitel 21, Massiv Parallele Programme
Parallelism in Imperative Languages

Predominant:

- **Data-parallel Languages** (e.g. High Performance Fortran)
- **Libraries (PVM, MPI)** \(\leadsto\) **Message Passing Model** (C, C++, Fortran)
Parallelism in Functional Languages

Predominant:

- Implicit (expression) parallelism
- Explicit parallelism
- Algorithmic skeletons
Implicit Parallelism

...also known as \textit{expression parallelism}.

Let \( f(e_1, \ldots, e_n) \) be a functional expression:

Then

\begin{itemize}
  \item Arguments (and functions) can be evaluated \textit{in parallel}.
  \item \textbf{Most important advantage:} Parallelism \textit{for free}! No effort for the programmer at all.
  \item \textbf{Most important disadvantage:} Results often unsatisfying; e.g. granularity, load distribution, etc. is not taken into account.
\end{itemize}

Summing up, \textit{expression parallelism} is

\begin{itemize}
  \item easy to detect (i.e., for the compiler) but \textbf{hard to fully exploit}.
\end{itemize}
Explicit Parallelism

By means of

- Introducing meta-statements (e.g. to control the data and load distribution, communication)
- Most important advantage: Often very good results thanks to explicit hands-on control of the programmer.
- Most important disadvantage: High programming effort and loss of functional elegance.
Algorithmic Skeletons

...a compromise between

- explicit imperative parallel programming
- implicit functional expression parallelism
In the following

We assume a scenario with

- Massively parallel systems
- Algorithmic skeletons
Massively Parallel Systems

...characterized by

- large number of processors with
  - local memory
  - communication by message exchange

- MIMD-Parallel Processor Architecture (Multiple Instruction/Multiple Data)

Here we restrict ourselves to:

- SPMD-Programming Style (Single Program/Multiple Data)
Algorithmic Skeletons

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
Implementation of Algorithmic Skeletons

...in functional languages

- by special higher-order functions
- with parallel implementation
- embedded in sequential languages

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 \texttt{map} on lists:

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map} \_ \ [\_] = [\_]
\]

\[
\text{map} \ f \ (x:xs) = (f \ x) : (\text{map} \ f \ xs)
\]

Observation:

- Applying \texttt{f} to a list element does not depend on other list elements.

Obviously:

- Dividing the list into sublists followed by parallel application of \texttt{map} to the sublists: parallelization pattern \texttt{Farm}. 
Parallel Map on Distributed Lists

Illustration:

\[
f \[a_1,...,a_k,a_{k+1},...,a_m,a_{m+1},...a_m\] \]

<table>
<thead>
<tr>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Computation</td>
</tr>
<tr>
<td>Composition</td>
</tr>
</tbody>
</table>

\[
[f \[a_1,...,a_k\] f \[a_{k+1},...,a_m\] f \[a_{m+1},...,a_m\] \]

\[
[b_1,...,b_k] \quad [b_{k+1},...,b_m] \quad [b_{m+1},...,b_m] \]

\[
[b_1,...,b_k, b_{k+1},...,b_m, b_{m+1},...b_m] \]

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

- so-called data-parallel skeletons.

Note the difference:

- **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, hence dynamically generated.
Programming of a Parallel Application

...using algorithmic skeletons:

- Recognizing problem-inherent parallelism.
- Selecting an adequate data distribution (granularity).
- Selecting a suitable skeleton from a library.
- Instantiating a problem-specific skeleton.

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 (investigated by various researchers).
...describe the

- decomposition and communication pattern of a data structure.
Illustration of a Simple List Cover

Distributing a list on 3 processors $p_0$, $p_1$, and $p_2$:

<table>
<thead>
<tr>
<th>a1</th>
<th>ak</th>
<th>ak+1</th>
<th>am</th>
<th>am+1</th>
<th>am</th>
</tr>
</thead>
</table>

$p_0$      $p_1$      $p_2$

Illustration of a List Cover with Overlapping Elements

General Cover Structure

Cover =

Type $S \ a$  -- Whole object
$C \ b$  -- Cover
$U \ c$  -- Local sub-objects

$\text{split} :: S \ a \rightarrow C \ (U \ a)$  -- Decomposing the
-- original object

$\text{glue} :: C \ (U \ a) \rightarrow S \ a$  -- Composing the
-- original object

It is required:

$\text{glue} \ . \ \text{split} = \text{id}$

Note: The above code snippet is not (valid) Haskell.
Implementation in a Programming Language

Implementing covers requires support for

- the specification of covers.
- the programming of algorithmic skeletons on covers.
- the provision of often used skeletons in libraries.

It is

- currently a hot research topic in functional programming.
Last but not least

Implementing skeletons

▷ by message passing via skeleton hierarchies.
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 17.2

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
- ...
“Real World” Haskell (2)

This support, which comes mostly in terms of

- sophisticated libraries

makes Haskell a reasonable choice for addressing and solving

- Real World Problems

since such a choice depends much on the ability and support a programming language (environment) provides for linking and connecting to the “outer world.”
Chapter 17.2: Further Reading (1)


Chapter 17.2: Further Reading (2)

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 (3)

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 Prospectives
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 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’
Contest Announcement at ICFP 2012 (1)

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.
Contest Announcement at ICFP 2012 (2)

Any programming language(s) may be used as long as the submitted program can be run by the judges on a standard Linux environment with no network connection. Details of the judges’ environment will be announced later.

There will be cash prizes for the first and second place teams, the team winning the lightning divison, 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.)

Contest Announcement at ICFP 2016

- This year the contest is going to take place from August 5, 2016 to August 8, 2016.
- Detailed information on it will be announced soon.
- Stay tuned for news on this year’s contest at http://conf.researchr.org/home/icfp-2016
- Programming Contest Chair: Gabriele Keller, UNSW, Sydney, Australien

Functional Programming in the Real World


- Haskell in Industry and Open Source: www.haskell.org/haskellwiki/Haskell_in_industry
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 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: 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
Bibliography
Reading

...for deepened and independent studies.

- I Textbooks
- II Monographs
- III Volumes
- IV Articles
- V Haskell 98 – Language Definition
- V The History of Haskell
I Textbooks (1)


I Textbooks (2)


I Textbooks (3)


I Textbooks (4)


I Textbooks (5)


I Textbooks (6)


Mark P. Jones, Alastair Reid et al. (Eds.). *The Hugs98 User Manual*. www.haskell.org/hugs


I Textbooks (7)


I Textbooks (8)


I Textbooks (9)


I Textbooks (10)


I Textbooks (11)


II Monographs (1)


II Monographs (2)


IV Articles (1)


IV Articles (2)


IV Articles (3)


IV Articles (4)


IV Articles (5)


IV Articles (6)


IV Articles (7)


IV Articles (8)


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)


www.curry-language.org/


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 (29)


IV Articles (32)


IV Articles (33)


IV Articles (34)


IV Articles (36)


IV Articles (37)


IV Articles (38)


IV Articles (39)


IV Articles (41)


IV Articles (42)


IV Articles (45)


IV Articles (46)


IV Articles (47)

V Haskell 98 – Language Definition


VI The History of Haskell


Appendix
Mathematical Foundations
A.1
Sets and Relations
Sets and Relations (1)

Definition (A.1.1)

Let $M$ be a set and $R$ a relation on $M$, i.e. $R \subseteq M \times M$.

Then $R$ is called

- reflexive iff $\forall m \in M. \ m R m$
- transitive iff $\forall m, n, p \in M. \ m R n \land n R p \Rightarrow m R p$
- anti-symmetric iff $\forall m, n \in M. \ m R n \land n R m \Rightarrow m = n$
Related notions (though less important for us here):

**Definition (A.1.2)**

Let $M$ be a set and $R \subseteq M \times M$ a relation on $M$. Then $R$ is called

- **symmetric** iff $\forall m, n \in M. \ m R n \iff n R m$
- **total** iff $\forall m, n \in M. \ m R n \lor n R m$
A.2

Partially Ordered Sets
Partially Ordered Sets

Definition (A.2.1, Quasi-Order, Partial Order)

A relation $R$ on $M$ is called a

- quasi-order iff $R$ is reflexive and transitive
- partial order iff $R$ is reflexive, transitive, and anti-symmetric

For the sake of completeness we recall:

Definition (A.2.2, Equivalence Relation)

A relation $R$ on $M$ is called an

- equivalence relation iff $R$ is reflexive, transitive, and symmetric
Remark

...a partial order is an anti-symmetric quasi-order, an equivalence relation a symmetric quasi-order.

Note: We here use terms like “partial order” as a short hand for the more accurate term “partially ordered set.”
Definition (A.2.3, Bounds, least/greatest Elements)

Let \((Q, \sqsubseteq)\) be a quasi-order, let \(q \in Q\) and \(Q' \subseteq Q\).

Then \(q\) is called

- **upper (lower) bound** of \(Q'\), in signs: \(Q' \sqsubseteq q\) (\(q \sqsubseteq Q'\)), if for all \(q' \in Q'\) holds: \(q' \sqsubseteq q\) (\(q \sqsubseteq q'\))

- **least upper (greatest lower) bound** of \(Q'\), if \(q\) is an upper (lower) bound of \(Q'\) and for every other upper (lower) bound \(\hat{q}\) of \(Q'\) holds: \(q \sqsubseteq \hat{q}\) (\(\hat{q} \sqsubseteq q\))

- **greatest (least) element** of \(Q\), if holds: \(Q \sqsubseteq q\) (\(q \sqsubseteq Q\))
Existence and Uniqueness of Bounds

We have:

- Given a partial order, least upper and greatest lower bounds are uniquely determined, if they exist.

- Given existence (and thus uniqueness), the least upper (greatest lower) bound of a set $P' \subseteq P$ of the basic set of a partial order $(P, \sqsubseteq)$ is denoted by $\bigcup P'$ ($\bigcap P'$). These elements are also called supremum and infimum of $P'$.

- Analogously this holds for least and greatest elements. Given existence, these elements are usually denoted by $\bot$ and $\top$. 
A.3

Lattices
Lattices and Complete Lattices

Definition (A.3.1, (Complete) Lattice)

Let \((P, \sqsubseteq)\) be a partial order. Then \((P, \sqsubseteq)\) is called a

- **lattice**, if each **finite** subset \(P'\) of \(P\) contains a least upper and a greatest lower bound in \(P\).

- **complete lattice**, if each subset \(P'\) of \(P\) contains a least upper and a greatest lower bound in \(P\).

**Hence:**

...(complete) lattices are special partial orders.
Properties of Complete Lattices

Lemma (A.3.2)

Let \((P, \sqsubseteq)\) be a complete lattice. Then we have:

1. \(\bot = \bigcup \emptyset = \bigcap P\) is the least element of \(P\).
2. \(\top = \bigcap \emptyset = \bigcup P\) is the greatest element of \(P\).

Lemma (A.3.3)

Let \((P, \sqsubseteq)\) be a partial order. Then the following claims 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 upper bound.
A.4

Complete Partially Ordered Sets
Complete Partial Orders

...a slightly weaker notion than a lattice that, however, is often sufficient in computer science and thus often a more adequate notion:

Definition (A.4.1, Complete Partial Order)

Let \((P, \sqsubseteq)\) be a partial order. Then \((P, \sqsubseteq)\) is called

- **complete**, or shorter a **CPO** (complete partial order), if each ascending chain \(C \subseteq P\) has a least upper bound in \(P\).
Remark

We have:

- A CPO \((C, \sqsubseteq)\) (more accurate would be: “chain-complete partially ordered set (CCPO)” ) has always a least element. This element is uniquely determined as the supremum of the empty chain and usually denoted by \(\bot\): \(\bot =_{df} \bigcup \emptyset\).
Chains

Definition (A.4.2, Chain)
Let \((P, \sqsubseteq)\) be a partial order.

A subset \(C \subseteq P\) is called

- chain of \(P\), if the elements of \(C\) are totally ordered. For \(C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}\) \((\{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\})\) we also speak more precisely of an ascending (descending) chain of \(P\).

A chain \(C\) is called

- finite, if \(C\) is finite; infinite otherwise.
Finite Chains, finite Elements

Definition (A.4.3, Chain-finite)
A partial order \((P, \sqsubseteq)\) is called

- chain-finite (German: kettenendlich) iff \(P\) does not contain infinite chains

Definition (A.4.4, Finite Elements)
An element \(p \in P\) is called

- finite iff the set \(Q = \{q \in P \mid q \sqsubseteq p\}\) is free of infinite chains

- finite relative to \(r \in P\) iff the set \(Q = \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}\) does not contain infinite chains
(Standard) CPO Constructions (1)

Flat CPOs.

Let \((C, \sqsubseteq)\) be a CPO. Then \((C, \sqsubseteq)\) is called flat, if for all \(c, d \in C\) holds:

\[
c \sqsubseteq d \iff c = \bot \lor c = d
\]

\[
\begin{array}{c}
c_1 \quad c_2 \quad c_3 \quad c_4 \quad c_5 \quad c_6 \quad c_7 \quad \ldots
\end{array}
\]
(Standard) CPO Constructions (2)

Product construction.

Let \((P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)\) be CPOs. Then

- the non-strict (direct) product \((\times P_i, \sqsubseteq)\) with
  
  \[
  \times P_i, \sqsubseteq = (P_1 \times P_2 \times \ldots \times P_n, \sqsubseteq) \quad \text{with} \quad \forall (p_1, p_2, \ldots, p_n), \quad (q_1, q_2, \ldots, q_n) \in \times P_i. \quad (p_1, p_2, \ldots, p_n) \sqsubseteq (q_1, q_2, \ldots, q_n) \iff \forall i \in \{1, \ldots, n\}. \quad p_i \sqsubseteq_i q_i
  \]

- and the strict (direct) product (smash product) with
  
  \[
  \otimes P_i, \sqsubseteq = (P_1 \otimes P_2 \otimes \ldots \otimes P_n, \sqsubseteq), \quad \text{where} \quad \sqsubseteq \text{ is defined as above under the additional constraint:} \]

\[
(p_1, p_2, \ldots, p_n) = \bot \iff \exists i \in \{1, \ldots, n\}. \quad p_i = \bot_i
\]

are CPOs, too.
Sum construction.

Let \((P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)\) CPOs. Then

- the direct sum \((\bigoplus P_i, \sqsubseteq)\) with
  \[(\bigoplus P_i, \sqsubseteq) = (P_1 \cup P_2 \cup \ldots \cup P_n, \sqsubseteq)\] is disjoint union of \(P_i, i \in \{1, \ldots, n\}\) and \(\forall p, q \in \bigoplus P_i. p \sqsubseteq q \iff \exists i \in \{1, \ldots, n\}. p, q \in P_i \land p \sqsubseteq_i q\)

is a CPO.

**Note:** The least elements of \((P_i, \sqsubseteq_i), i \in \{1, \ldots, n\}\), are usually identified, i.e., \(\bot =_{df} \bot_i, i \in \{1, \ldots, n\}\)
(Standard) CPO Constructions (4)

Function-space construction.

Let \((C, \sqsubseteq_C)\) and \((D, \sqsubseteq_D)\) be two CPOs and \([C \to D] = \{ f : C \to D \mid f \text{ continuous} \}\) the set of continuous functions from \(C\) to \(D\).

Then

- the continuous function space \(([C \to D], \sqsubseteq)\) is a CPO where
  - \(\forall f, g \in [C \to D]. \ f \sqsubseteq g \iff \forall c \in C. \ f(c) \sqsubseteq_D g(c)\)
Monotonic, Continuous Functions on CPOs

**Definition (A.4.5, Monotonic, Continuous Function)**

Let \((C, \sqsubseteq_C)\) and \((D, \sqsubseteq_D)\) be two CPOs and let \(f : C \rightarrow D\) be a function from \(C\) to \(D\).

Then \(f\) is called

- **monotonic** iff \(\forall c, c' \in C. \quad c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')\)
  
  (Preservation of the ordering of elements)

- **continuous** iff \(\forall C' \subseteq C. \quad f(\bigsqcup_C C') = \bigsqcup_D f(C')\)
  
  (Preservation of least upper bounds)
Properties

Using the notations introduced before, we have:

Lemma (A.4.6)

\[ f \text{ is monotonic iff } \forall C' \subseteq C. \quad f(\bigcup_C C') \supseteq_D \bigcup_D f(C') \]

Corollary (A.4.7)

A continuous function is always monotonic, i.e. \( f \text{ continuous implies } f \text{ monotonic.} \)
Inflationary Functions on CPOs

Definition (A.4.8, Inflationary Function)

Let \((C, \sqsubseteq)\) be a CPO and let \(f : C \rightarrow C\) be a function on \(C\). Then \(f\) is called

- inflationary (increasing) iff \(\forall c \in C. \ c \sqsubseteq f(c)\)
A.5

Fixed Point Theorems
Least and Greatest Fixed Points

Definition (A.5.1, (Least/Greatest) Fixed Point)

Let $(C,\sqsubseteq)$ be a CPO, $f : C \to C$ be a function on $C$ and let $c$ be an element of $C$, i.e., $c \in C$.

Then $c$ is called

- fixed point of $f$ iff $f(c) = c$

A fixed point $c$ of $f$ is called

- least fixed point of $f$ iff $\forall d \in C. f(d) = d \Rightarrow c \sqsubseteq d$
- greatest fixed point of $f$ iff $\forall d \in C. f(d) = d \Rightarrow d \sqsubseteq c$

Notation:

- The least resp. greatest fixed point of a function $f$ is usually denoted by $\mu f$ resp. $\nu f$. 
Conditional Fixed Points (2)

Definition (A.5.2, Conditional Fixed Point)

Let \((C, \sqsubseteq)\) be a CPO, \(f : C \to C\) be a function on \(C\) and let \(d, c_d \in C\).

Then \(c_d\) is called

- conditional (German: bedingter) least fixed point of \(f\) wrt \(d\) iff \(c_d\) is the least fixed point of \(C\) with \(d \sqsubseteq c_d\), i.e. for all other fixed points \(x\) of \(f\) with \(d \sqsubseteq x\) holds: \(c_d \sqsubseteq x\).
Fixed Point Theorem

Theorem (A.5.3, Knaster/Tarski, Kleene)

Let \((C, \sqsubseteq)\) be a CPO and let \(f : C \to C\) be a continuous function on \(C\).

Then \(f\) has a least fixed point \(\mu f\), which equals the least upper bound of the chain (so-called Kleene-Chain) \(\{\bot, f(\bot), f^2(\bot), \ldots\}\), i.e.

\[
\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup \{\bot, f(\bot), f^2(\bot), \ldots\}
\]
Proof of Fixed Point Theorem A.5.3 (1)

We have to prove:

\[ \mu f \]

1. exists
2. is a fixed point
3. is the least fixed point

of \( f \).
Proof of Fixed Point Theorem A.5.3 (2)

1. Existence

- It holds \(f^0 ⊥ = df ⊥\) and \(⊥ ⊑ c\) for all \(c ∈ C\).
- By means of (natural) induction we can show: \(f^n ⊑ f^nc\) for all \(c ∈ C\).
- Thus we have \(f^n ⊑ f^m ⊥\) for all \(n, m\) with \(n ≤ m\). Hence, \(\{f^n ⊥ \mid n ≥ 0\}\) is a (non-finite) chain of \(C\).
- The existence of \(\bigsqcup_{i ∈ \mathbb{N}_0} f^i(⊥)\) is thus an immediate consequence of the CPO properties of \((C, ⊑)\).
Proof of Fixed Point Theorem A.5.3 (3)

2. Fixed point property

\[ f( \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)) = \bigsqcup_{i \in \mathbb{N}_0} f(f^i \bot) = \bigsqcup_{i \in \mathbb{N}_1} f^i \bot \]

(K chain \Rightarrow \bigsqcup K = \bot \cup \bigsqcup K) = (\bigsqcup_{i \in \mathbb{N}_1} f^i \bot) \cup \bot

(f^0 \bot = \bot) = \bigsqcup_{i \in \mathbb{N}_0} f^i \bot \]
Proof of Fixed Point Theorem A.5.3 (4)

3. Least fixed point

- Let $c$ be an arbitrarily chosen fixed point of $f$. Then we have $\bot \sqsubseteq c$, and hence also $f^n \bot \sqsubseteq f^n c$ for all $n \geq 0$.
- Thus, we have $f^n \bot \sqsubseteq c$ because of our choice of $c$ as fixed point of $f$.
- Thus, we also have that $c$ is an upper bound of $\{f^i(\bot) \mid i \in \mathbb{N}_0\}$.
- Since $\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)$ is the least upper bound of this chain by definition, we obtain as desired $\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) \sqsubseteq c$. □
Conditional Fixed Points

Theorem (A.5.4, Conditional Fixed Points)
Let \((C, \sqsubseteq)\) be a CPO, let \(f : C \to C\) be a continuous, inflationary function on \(C\), and let \(d \in C\).

Then \(f\) has a unique conditional fixed point \(\mu f_d\). This fixed point equals the least upper bound of the chain \(\{d, f(d), f^2(d), \ldots\}\), i.e.

\[
\mu f_d = \bigsqcup_{i \in \IN_0} f^i(d) = \bigsqcup_{i \in \IN_0} \{d, f(d), f^2(d), \ldots\}
\]
Finite Fixed Points

Theorem (A.5.5, Finite Fixed Points)

Let \((C, \sqsubseteq)\) be a CPO and let \(f : C \to C\) be a continuous function on \(C\).

Then we have: If two elements in a row occurring in the Kleene-chain of \(f\) are equal, e.g. \(f^i(\bot) = f^{i+1}(\bot)\), then we have: \(\mu f = f^i(\bot)\).
Existence of Finite Fixed Points

Sufficient conditions for the existence of finite fixed points e.g. are

- Finiteness of domain and range of $f$
- $f$ is of the form $f(c) = c \sqcup g(c)$ for monotone $g$ on some chain-complete domain
A.6

Cones and Ideals
Cones und Ideals

Definition (A.6.1, Directed Set, Cone, Ideal)
Let \((P, \sqsubseteq)\) be a partial order and \(Q\) be a subset of \(P\), i.e., \(Q \subseteq P\).

Then \(Q\) is called
- **directed set** (German: *gerichtet (gerichtete Menge)*), if each *finite* subset \(R \subseteq Q\) has a supremum in \(Q\), i.e. \(\exists q \in Q. \, q = \bigsqcup R\)
- **cone** (German: *Kegel*), if \(Q\) is downward closed, i.e. \(\forall q \in Q \, \forall p \in P. \, p \sqsubseteq q \Rightarrow p \in Q\)
- **ideal** (German: *Ideal*), if \(Q\) is a directed cone, i.e. if \(Q\) is downward closed and each finite subset has a supremum in \(Q\).

**Note:** If \(Q\) is a directed set, then, we have because of \(\emptyset \subseteq Q\) also \(\bigsqcup \emptyset = \bot \in Q\) and thus \(Q \neq \emptyset\).
Completion of Ideals

Theorem (A.6.2, Completion of Ideals)

Let \((P, \subseteq)\) be a partial order and let \(I_P\) be the set of all ideals of \(P\). Then we have:

\( (I_P, \subseteq) \) is a CPO.

Induced “completion:”

Identifying each element \(p \in P\) with its corresponding ideal \(I_p = \{ q \mid q \subseteq p \}\) yields an embedding of \(P\) into \(I_P\) with \(p \subseteq q \iff I_P \subseteq I_Q\)

Corollary (A.6.3, Extensibility of Functions)

Let \((P, \subseteq_P)\) be a partial order and let \((C, \subseteq_C)\) be a CPO. Then we have: All monotonic functions \(f : P \to C\) can be extended to a uniquely determined continuous function \(\hat{f} : I_P \to C\).
Summing up

The preceding result implies:

- Streams constitute a CPO.
- Recursive equations and functions on streams are well-defined.
- The application of a function to the finite prefixes of a stream yields the chain of approximations of the application of the function to the stream itself; it is thus correct.
Appendix A: Further Reading (1)


Appendix A: Further Reading (2)


Appendix A: Further Reading (3)

Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Chapter 5.1, Ordnungsrelationen; Chapter 5.2, Ordnungen und Teilstrukturen)


Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about Programs; Chapter 17.9, Proof revisited)