Fortgeschrittene funktionale Programmierung

LVA 185.A05, VU 2.0, ECTS 3.0
SS 2019

(Stand: 26.06.2019)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Table of Contents

Table of Contents (1)

Table of Contents (2)

Table of Contents (3)

Part III: Quality Assurance

Table of Contents (4)

Table of Contents (5)

Part IV: Advanced Language Concepts

Table of Contents (6)

Table of Contents (7)

Table of Contents (8)

Table of Contents (9)

Table of Contents (10)

Part V: Applications

Table of Contents (11)

Table of Contents (12)

Table of Contents (13)

Table of Contents (14)

Part VI: Extensions, Perspectives

Appendix

Table of Contents (15)

Table of Contents (16)

Table of Contents (17)

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.
  Communications of the ACM 54(11):53-58, 2011.

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.
  Computer Journal 32(2):98-107, 1989.

Chapter 1

Why Functional Programming Matters

Why Functional Programming Matters

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

- Computer Journal 32(2):98-107, 1989.
- Research Topics in Functional Programming. David Turner (Ed.), Addison-Wesley, 1990.
- http://www.cs.chalmers.se/∼rjmh/Papers/whyfp.html

"...an attempt to demonstrate to the "real world" that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by ma- king it clear what those advantages are."

Chapter 1.1

Setting the Stage

Introductory Statement

A matter of fact:

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

Claim:

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

Purpose of the position statement:

- Providing evidence for this claim.

A First Observation

...functional programming owes its name the fact that programs are composed of only functions:

- The main program is itself a function.
- It accepts the program's input as its arguments and delivers the program's output as its result.
- It is defined in terms of other functions, which themselves are defined in terms of still more functions (eventually by primitive functions).

Evidence by Folk Knowledge: Soft Facts

...characteristics & advantages of functional programming:

Functional programs are

- free of assignments and side-effects
- function calls have no effect except of computing their result

⇒ functional programs are thus free of a major source of bugs

- the evaluation order of expressions is irrelevant, expressions can be evaluated any time
- programmers are free from specifying the control flow explicitly
- expressions can be replaced by their value and vice versa; programs are referentially transparent

⇒ functional programs are thus easier to cope with mathematically (e.g., for proving their correctness)

Note

...this 'folk knowledge' list of characteristics and advantages of functional programming is essentially a negative 'is-not' characterization:

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

Evidence by Folk Knowledge: Hard(er) Facts

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

Functional programs are

- a magnitude of order smaller than conventional programs

$\Rightarrow$ functional programmers are thus much more productive

Issue left open, however:

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

Hardly. Dropping features reminds more to a medieval monk denying himself the pleasures of life in the hope of getting virtuous.

Summing up

The ‘folk knowledge’ catalogue is not satisfying; in particular, it does not provide

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

We need a positive characterization of the vital nature of

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

A Striking Analogue

...structured vs. non-structured programming.

Structured programs are

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

$\Rightarrow$ easier to mathematically cope with than unstructured programs

...this is also essentially a negative 'is-not' characterization.

Conceptually more Important

Structured programs are

- designed modularly

in contrast to non-structured ones.

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

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

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

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

Key Thesis of John Hughes

The expressiveness

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

Functional programming

- provides two new, especially powerful glues:

  1. Higher-order functions
  2. Lazy evaluation

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

Modularization

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

In the following

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

Utilizing

- higher-order functions to glueing functions together
- lazy evaluation to glueing programs together

Chapter 1.2

Glueing Functions Together

Glueing Functions Together

Syntax (in the flavour of Miranda^{TM}):

- Lists

```
listof X ::= nil | cons X (listof X)
```

- Abbreviations (for convenience)

```
[]        means  nil
[1]       means  cons 1 nil
[1,2,3]   means  cons 1 (cons 2 (cons 3 nil)))
```

Example: Adding the elements of a list

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

```
sum = reduce add 0
      where add x y = x+y
```

This reveals the definition of the higher-order function `reduce` almost immediately:

```
(reduce f x) nil          = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Recall

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

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

Immediate Benefit: Re-use of the HoF `reduce`

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

2. Multiplying the elements of a list

```
product = reduce mult 1
          where mult x y = x*y
```

3. Test, if *some* element of a list equals "true"

```
anytrue = reduce or false
```

4. Test, if *all* elements of a list equal "true"

```
alltrue = reduce and true
```

5. Concatenating two lists

```
append a b = reduce cons b a
```

6. Doubling each element of a list

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

How does it work? (1)

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

- `cons` are replaced by `f`
- `nil` by `a`

in list values.

Examples:

```
1) Addition:
   reduce add 0 (cons 2 (cons 3 (cons 5 nil)))
            ->> (add  2 (add  3 (add  5 0  )))
            ->> 10

2) Multiplication:
   reduce mult 1 (cons 2 (cons 3 (cons 5 nil)))
             ->> (mult 2 (mult 3 (mult 5 1  )))
             ->> 30
```

How does it work? (2)

Examples (cont'd):

5) Concatenating two lists

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

```
 append a b = reduce cons b a

append [1,2] [3,4]
 ->> reduce cons [3,4] [1,2]
 ->> (reduce cons [3,4]) (cons 1 (cons 2 nil))
 ->> { replacing cons by cons and nil by [3,4] }
                          (cons 1 (cons 2 [3,4]))
 ->> { expanding [3,4] }
     (cons 1 (cons 2 (cons 3 (cons 4 nil))))
 ->> { compressing the list expression }
     [1,2,3,4]
```

How does it work? (3)

Examples (cont'd):

6) Doubling each element of a list

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

Note that doubleandcons can be modularized further:

- First step

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

- Second step

```
fandcons f = cons . f
```

where "." denotes sequential composition of functions:

```
(f . g) h = f (g h)
```

How does it work? (4)

...correctness of the two modularization steps follows from

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

which yields as desired:

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

How does it work? (5)

Putting 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
```

where

```
map f = reduce (cons . f) nil
```

i.e., `map` applies a function `f` to every element of a list.

Homework

Using the functions introduced so far, we can define:

- Adding the elements of a matrix

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

Think about how summatrix works.

Summing up

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

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

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

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

Generalization

...to more complex data structures:

Example: Trees

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

A value of type (treeof X):

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

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

The Higher-order Function redtree

Analogously to reduce on lists we introduce a higher-order function redtree on trees:

```
redtree f g a (node label subtrees)
 = f label (redtree′ f g a subtrees)
where
 redtree′ f g a (cons subtree rest)
  = g (redtree f g a subtree) (redtree′ f g a rest)
 redtree′ f g a nil = a
```

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

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

in tree values.

Applications (1)

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

Running Example:

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

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

Applications (2)

1. Adding the labels of the leaves of a tree

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

Example:

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

```
sumtree (node 1
          (cons (node 2 nil)
                (cons (node 3 (cons (node 4 nil) nil))
                nil)))
    ->> (add  1
          (add  (add  2 0  )
                (add  (add  3 (add  (add  4 0  ) 0  ))
                0  )))
    ->> 10
```

Applications (3)

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

```
labels = redtree cons append nil
```

Example:

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

```
sumtree (node 1
          (cons  (node 2 nil)
                 (cons (node 3 (cons (node 4 nil) nil))
                 nil)))
    ->> (cons 1
          (app'd (cons 2 nil)
                 (app'd (cons 3 (app'd (cons 4 nil) nil))
                 nil)))
    ->> [1,2,3,4]
```

Applications (4)

3. A function `maptree` applying a function `f` to every label of a tree

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

Example: Homework.

Summing up (1)

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

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

Once the higher-order function is implemented, lots of

- further functions can be implemented essentially without any further effort!

Summing up (2)

Lesson learnt:

- Whenever a new data type is introduced, implement first a higher-order function allowing to process values of this type (e.g., visiting each component of a structured data value such as nodes in a graph or tree).

Benefits:

- Manipulating elements of this data type becomes easy; knowledge about this data type is locally concentrated and encapsulated.

Look&feel:

- Whenever a new data structure demands a new control structure, then this control structure can easily be added following the methodology used above (to some extent this resembles the concepts known from conventional extensible languages).

Chapter 1.3

Glueing Programs Together

Glueing Programs Together

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

- If `f` and `g` are complete functional programs, then also their composition

  ```
  (g . f)
  ```

  is a program. Applied to `in` as input, it yields the output

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

- A possible implementation using conventional glue is:
  $\rightsquigarrow$ Communication via files

  Possible problems:

  - Temporary files used for communication can be too large
  - `f` might not terminate

Functional Glue

...lazy evaluation allows a more elegant approach.

This is to decompose a program into a

- generator
- selector

component/module, which are then glued together.

Intuitively:

- The generator "runs as little as possible" until it is terminated by the selector.

Three Examples

...for illustrating this modularization strategy:

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

Example 1: Square Root Computation

...according to Newton-Raphson.

Given: `N`

Sought: `squareRoot(N)`

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

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

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

I.e., `a` stores the value of the square root of `N`.

We consider first

...a typical imperative (Fortran) implementation:

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

⇝ essentially a monolithic, not decomposable program.

The Functional Version: New Approximations

Computing the next approximation from the previous one:

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

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

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

The Functional Version: Writing a Generator

The function `repeat` computes this (possibly infinite) sequence of approximations. It is the generator component in this example:

Generator A:

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

Applying `repeat` to the arguments `next N` and `a0` yields the desired sequence of approximations:

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

The Functional Version: Writing a Selector

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

Remedy: Taming the generator by a selector to compute `squareroot N` only up to a given tolerance $\text{eps} > 0$:

Selector A:

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

The Functional Version: Combining Gen./Sel.

...to obtain the final program.

Composition: Glueing generator and selector together:

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

$\rightsquigarrow$ We are done!

The Functional Version: Summing up

- `repeat`: generator program/module:
  `[a0, g a0, g(g a0), g(g(g a0)),...]`
  ...potentially infinite, no pre-defined limit of length.
- `within`: selector program/module:
  $\texttt{g}^i$ `a0` with `abs(`$\texttt{g}^i$ `a0 -` $\texttt{g}^{i+1}$ `a0) <= eps`
  ...lazy evaluation ensures that the selector function is applied eventually $\Rightarrow$ termination!

Note: Lazy evaluation ensures that both programs/modules (generator and selector) run strictly synchronized.

Re-using Modules

Next, we will show that

- generators
- selectors

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

We are going to start with re-using the generator module.

Re-using a Generator w/ new Selectors

Consider a new criterion for termination:

- Instead of awaiting the difference of successive approximations to approach zero (`i.e., <= eps`), await their ratio to approach one (`i.e., <= 1+eps`)

Selector B:

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

Composition: Glueing old generator and new selector together:

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

⇝ We are done!

Next: Re-using a Selector w/ new Generators

Note that the module generator in the previous example, i.e.

- the component computing the sequence of approximations

has been re-used unchanged.

Next, we will re-use the two selector modules considering two examples:

- Numerical integration
- Numerical differentiation

Example 2: Numerical Integration

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

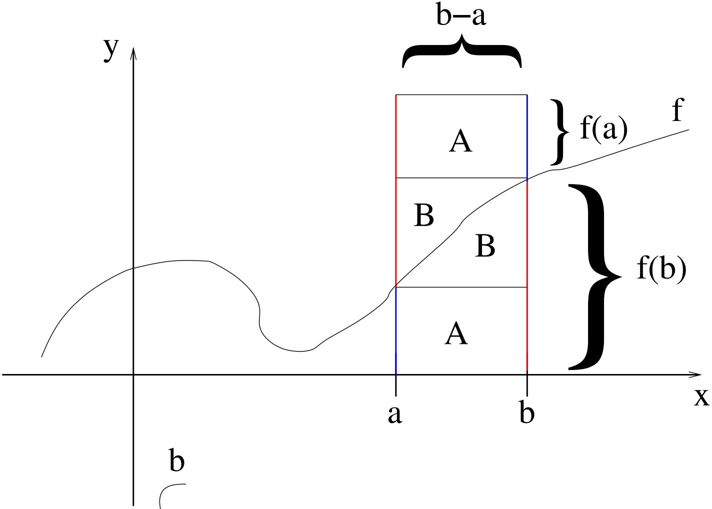
Sought: The area under `f` between `a` and `b`

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

```
easyintegrate f a b = (f a + f b) * (b-a) / 2
```

This is sufficiently precise, however, at most for very small intervals.

Illustration



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

Writing a Generator

Strategy

- Halve the interval, compute the areas for both sub-intervals according to the previous formula, and add the two results
- Continue the previous step repeatedly

The function `integrate` realizes this strategy:

Generator B:

```
integrate f a b
  = cons (easyintegrate f a b)
         map addpair (zip (integrate f a mid)
                          (integrate f mid b)))
    where mid = (a+b)/2
```

where

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```

Re-using Selectors w/ the new Generator

Note, evaluating the new generator term `integrate f a b` does not terminate!

Remedy: Taming the new generator by the previously defined two selectors to compute `integrate f a b` only up to some given limit `eps` > 0.

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

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

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

Summing up

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

Note, the two selector modules

- picking the solution from the stream of approximate solutions

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

- Lazy evaluation is the key to synchronize the generator and selector modules!

A Note on Efficiency

The generator `integrate` as defined previously is

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

Introducing locally defined values as shown below removes this deficiency:

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

Example 3: Numerical Differentiation

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

Sought: The slope of `f` at point `x`

Naive Implementation:

...supposed that the function `f` between `x` and `x+h` does not "curve much"

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

This is sufficiently precise, however, at most for very small values of `h`.

Writing a Generator/Selector Combination

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

Generator C:

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

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

Selector A:

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

Homework: Combine Generator C with Selector B, too.

Summing up

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

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

This composition/modularization principle can be further generalized to combining

- generators, selectors, filters, and transformers

as illustrated in Chapter 2.

Chapter 1.4

Summary

Findings (1)

Central Thesis of John Hughes

- Modularity is the key to programming in the large.

Observation

- Just modules (i.e., the capability of decomposing a problem) do not suffice.
- The benefit of modularly decomposing a problem into subproblems depends much on the capabilities for glueing the modules together.
- The availability of proper glue is essential!

Findings (2)

Facts

- Functional programming offers two new kinds of glue:
  - Higher-order functions (glueing functions)
  - Lazy evaluation (glueing programs)
- Higher-order functions and lazy evaluation allow substantially new exciting modular decompositions of problems (by offering elegant composition means) as here given evidence by an array of simple, yet impressive examples
- In essence, it is the superior glue, which allows functional programs to be written so concisely and elegantly (rather than the freedom of assignments, etc.)

Findings (3)

Guidelines

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

  are the tools for achieving this!

The Question of Lazy or Eager Evaluation

...reconsidered. The final conclusion of John Hughes is:

- The benefits of lazy evaluation as a glue are so evident that lazy evaluation is too important to make it a second-class citizen.
- Lazy evaluation is possibly the most powerful glue functional programming has to offer.
- Access to such a powerful means should not airily be dropped.

Lasst uns faul in allen Sachen,
[...]
nur nicht faul zur Faulheit sein.

Gotthold Ephraim Lessing (1729-1781)
dt. Dichter und Dramatiker

Looking ahead

…in Chapter 2 and Chapter 3 we will discuss the power higher-order functions and lazy evaluation provide the programmer with in further detail:

- Stream programming: thanks to lazy evaluation.
- Algorithm patterns: thanks to higher-order functions.

Chapter 1.5

References, Further Reading

Chapter 1: Further Reading (1)

- Stephen Chang, Matthias Felleisen. *The Call-by-Need Lambda Calculus, Revisited*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 128-147, 2012.
- Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. Communications of the ACM 21(3):359-411, 1989.
- John Hughes. *Why Functional Programming Matters*. Computer Journal 32(2):98-107, 1989.
- Mark P. Jones. *Functional Thinking*. Lecture at the 6th International Summer School on Advanced Functional Programming, Boxmeer, The Netherlands, 2008.

Chapter 1: Further Reading (2)

- Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
- Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the Occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL 2003), 2003. `http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/`

Chapter 1: Further Reading (3)

- Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*. Susan Eisenbach (Ed.), Ellis Horwood, 7-8, 1987.
- Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.
- John Hughes. Why Functional Programming Matters. Eingeladener Hauptvortrag, Bangalore, 2016. `https://www.youtube.com/watch?v=XrNdvWqxBvA`

Part II

Programming Principles

Chapter 2

Programming with Streams

Motivation

Streams $\widehat{=}$ Infinite Lists $\widehat{=}$ Lazy Lists: ...used synonymously.

Programming with streams

- Applications
  - Streams plus lazy evaluation yield new modularization principles
    - Generator/selector
    - Generator/filter
    - Generator/transformer

    as instances of the Generator/Prune Paradigm
  - Pitfalls and remedies
- Foundations
  - Well-definedness of functions on streams (cf. Appendix A.7.5)
  - Proving properties of programs with streams (cf. Chapter 6.3.4, 6.4, 6.5, 6.6)

Chapter 2.1

Streams

Streams

...combined with lazy evaluation often

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

More on this next.

Streams

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

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

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

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

This way

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

Simple Examples of Streams

- Built-in streams in Haskell

```
[0..]   ->> [0,1,2,3,4,5,...
[0,2..] ->> [0,2,4,6,8,10,...
[1,3..] ->> [1,3,5,7,9,11,...
[1,1..] ->> [1,1,1,1,1,1,...
```

- User-defined streams in Haskell

```
ones = 1 : ones
```

Illustration

```
 ones ->> 1 : ones
      ->> 1 : (1 : ones)
      ->> 1 : (1 : (1 : ones))
      ->> ...
```

`ones` represents an infinite list (or a stream).

Corecursive Definitions

Definitions of the form

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

defining the streams of "ones," "twos," and "threes"

- are called corecursive.

Corecursive definitions

- look like recursive definitions but lack a base case.
- always yield infinite objects.
- remind to Münchhausen's famous trick of "sich am eigenen Schopfe aus dem Sumpf zu ziehen"!

More Streams defined corecursively

- The stream of natural numbers `nats`

```
nats = 0 : map (+1) nats
 ->> [0,1,2,3,...
```

- The stream of even natural numbers `evens`

```
evens = 0 : map (+2) evens
 ->> [0,2,4,6,...
```

- The stream of odd natural numbers `odds`

```
odds = 1 : map (+2) odds
 ->> [1,3,5,7,...
```

- The stream of natural numbers `theNats`

```
theNats = 0 : zipWith (+) ones theNats
 ->> [0,1,2,3,...
```

Streams by List Comprehension and Recursion

- The stream of powers of an integer

```
powers :: Int -> [Int]
powers n = [n^x | x <- [0..]]

 ->> [1,n,n*n,n*n*n,...
```

- The stream of 'function applications,' the prelude function iterate

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

 ->> [x, f x, f (f x), f (f (f x)),...
```

- Application: Redefining powers in terms of iterate

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

More Applications of iterate

```
ones    = iterate id 1
twos    = iterate id 2
threes  = iterate id 3
nats    = iterate (+1) 0
theNats = iterate (+1) 0
evens   = iterate (+2) 0
odds    = iterate (+2) 1
powers  = iterate (*n) 1
```

Functions on Streams

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

Application: Generator/Selector pattern

$$\underbrace{\texttt{head}}_{\textit{Selector}}\quad \underbrace{\texttt{twos}}_{\textit{Generator}}\quad \texttt{->> head (2 : twos) ->> 2}$$

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

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

Recall

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

Example: Let `ignore` be defined by

```
ignore :: a -> b -> b
ignore a b = b
```

The leftmost-outermost operation of the term(s)

`ignore twos 42` $\widehat{=}$ ``twos `ignore` 42``

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

"...whenever there is a terminating reduction sequence of an expression, then normal order reduction will terminate."
Church/Rosser Theorem 12.3.2 (LVA 185.A03 FP)

More Functions on Streams

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

Application: Generator/Selector pattern

$\underbrace{\texttt{addFirstTwo}}_{\textit{Selector}}\ \underbrace{\texttt{twos}}_{\textit{Generator}}$ `->> addFirstTwo (2:twos)`

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

Functions yielding Streams

- User-defined stream-yielding functions

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

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

Applications

```
from 42       ->> [42,43,44,...

fromStep 3 2 ->> 3 : fromStep 5 2
             ->> 3 : 5 : fromStep 7 2
             ->> 3 : 5 : 7 : fromStep 9 2
             ->> ...
             ->> [3,5,7,9,11,13,15,...
```

- The stream `primes` of prime numbers...

Primes: The Sieve of Eratosthenes (1)

Intuition

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

Illustration

```
Step 1:
 2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17...
Step 2 ("with 2"):
 2  3     5     7     9     11    13    15    17...
Step 2 ("with 3"):
 2  3     5     7           11    13          17...
Step 2 ("with 5"):
 2  3     5     7           11    13          17...
...
```

Primes: The Sieve of Eratosthenes (2)

The stream of prime numbers `primes` (generator pattern):

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

$\underbrace{\texttt{primes}}_{\textit{Generator}}$ $\underbrace{\texttt{sieve}}_{\textit{Generator}}$

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

$\underbrace{\texttt{sieve}}_{\textit{Generator}}$

Primes: The Sieve of Eratosthenes (3)

Illustrating the generator property by stepwise evaluation:

```
primes
 ->> sieve [2..]
 ->> 2 : sieve [ y | y <- [3..], mod y 2 > 0]
 ->> 2 : sieve (3 : [ y | y <- [4..], mod y 2 > 0]
 ->> 2 : 3 : sieve [ z | z <- [ y | y <- [4..],
                          mod y 2 > 0 ],
                          mod z 3 > 0]
 ->> ...
 ->> 2 : 3 : sieve [ z | z <- [5, 7, 9..],
                          mod z 3 > 0]
 ->> ...
 ->> 2 : 3 : sieve [5,7,11,...
 ->> ...
 ->> [2,3,5,7,11,13,17,19,...
```

On Pitfalls in Applications with Streams

Implementing a prime number test (naively):

Consider

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

as a transforming selector (a-value to Bool-value).

Then

- `member primes 7 ->> True` ...as expected!
  t. Selector: ...working properly!
- `member primes 8 ->> ...` ...does not terminate!
  t. Selector: ...failing!

Homework: Why does the generator/transf. selector implementation of member and primes fail? How can the transf. selector member be modified to work properly as a transf. selector?

Generating (Pseudo) Random Numbers

Generating a sequence of (pseudo) random numbers:

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

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

Choosing

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

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

```
[17489, 59134, 9327, 52468, 43805, 8378,...
```

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

Generator/Transformer Modularization

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

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

```
scale :: Float -> Float -> [Int] -> [Float]
scale p q randSeq = map (f p q) randSeq
 where f :: Float -> Float -> Int -> Float
       f p q n = p + ((n * (q-p)) / (modulus-1))
```

Application: Generator/Transformer pattern

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

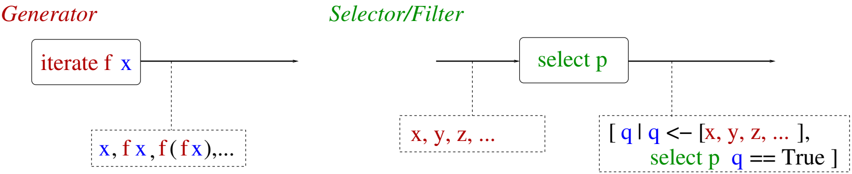
Principles of Modularization

...related to streams:

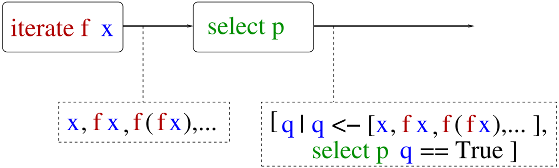
- The Generator/Selector Principle
  ...e.g., computing the square root, the *n*-th Fibonacci number
- The Generator/Filter Principle
  ...e.g., computing all even Fibonacci numbers
- The Generator/Transformer Principle
  ...e.g., "scaling" random numbers
- Further combinations of generators, filters, and selectors

The Generator/Sel./Filt. Modulariz. Principle

...at a glance:

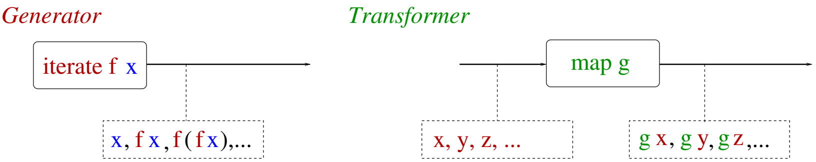


Linking Generator and Selector/Filter together

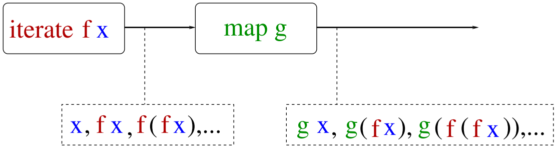


The Generator/Transf. Modulariz. Principle

...at a glance:



Linking Generator and Transformer together



The Fibonacci Numbers (1)

Recall: The stream of Fibonacci Numbers

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...`

relying on the function

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

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

The Fibonacci Numbers (2)

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

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

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

- inacceptably inefficient and slow!

The Fibonacci Numbers (3)

Illustration: By stepwise evaluation

```
fib 0 ->>  0                         --  1 call of fib

fib 1 ->>  1                         --  1 call of fib

fib 2 ->>  fib 1 + fib 0
      ->>  1 + 0
      ->>  1                        --  3 calls of fib

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

The Fibonacci Numbers (4)

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

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

The Fibonacci Numbers (5)

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

...tree-like recursion (with exponential growth!).

Recall (LVA 185.A03 FP): Complexity (1)

For further details, refer to:

- Peter Pepper. Funktionale Programmierung in OPAL, ML, Haskell und Gofer. 2nd Ed. (In German), 2003, Chapter 11.

$\mathcal{O}$ Notation

Let $f : \alpha \to IR^+$ be a function with some data type α 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:

$$\mathcal{O}(f) =_{df} \{h \mid h(n) \leq c * f(n) \text{ for some positive constant } c \text{ and all } n \geq N_0\}$$

Complexity (2)

...important cost functions:

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

Complexity (3)

...the impact of growing inputs in practice:

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

Streams to the Rescue

Stream programming

- can (often) help to conquer complexity!

The Stream of Fibonacci Numbers: Efficiently

Idea

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

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

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

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

Applications: Generator/Selector Pattern

Generator

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

Generator/Selector

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

where

```
take :: Int -> [a] -> [a]
take 0 _              = []
take _ []             = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _              = error "Negative argument"
```

From Stream `fibs` to Function `fib`

...the corecursive definition of the stream `fibs` suggests a conceptually new implementation of the Fibonacci function `fibs`:

$$\texttt{fib :: Int -> Int}$$
$$\texttt{fib n =} \underbrace{\texttt{last}}_{\textit{Selector 2}} \underbrace{\texttt{(take n}}_{\textit{Selector 1}} \underbrace{\texttt{fibs)}}_{\textit{Generator}}$$

Even shorter with only one selector:

$$\texttt{fib :: Int -> Int}$$
$$\texttt{fib n =} \underbrace{\texttt{fibs}}_{\textit{Generator}} \underbrace{\texttt{!! (n-1)}}_{\textit{Selector}}$$

Note the application of the generator/selector modularization in these two examples.

Lazy Evaluation is Essential for Performance

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

```
fibs
 ->> {Replace the call of fibs by the body of fibs}
       0 : 1 : add fibs (tail fibs)
 ->> {Replace both calls of fibs by the body of fibs}
       0 : 1 : add (0 : 1 : add fibs (tail fibs))
               (tail (0 : 1 : add fibs (tail fibs)))
 ->> {Application of tail}
       0 : 1 : add (0 : 1 : add fibs (tail fibs))
                        (1 : add fibs (tail fibs))
 ->> ... exponential effort!
```

...lazy evaluation ensures that common subexpressions (here, `tail` and `fibs`) are not computed multiple times!

Sharing: The Benefit of Lazy Evaluation (1)

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

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

     ->> { Introducing abbreviations allows sharing }
         0 : tf
         where tf = 1 : tf2 -- tf2 reminds to "tail
                               -- of tail of fibs"
                   where tf2 = add fibs tf

     ->> { Unfolding of add }
         0 : tf
         where tf = 1 : tf2
                   where tf2 = 1 : add tf tf2
```

Sharing: The Benefit of Lazy Evaluation (2)

```
->> {Repeating the above steps}
    0 : tf
    where tf = 1 : tf2
               where tf2 = 1 : tf3  (tf3 reminds to
                    "tail of tail  of tail of fibs")
                     where tf3 = add tf tf2
->> 0 : tf
    where tf = 1 : tf2
               where tf2 = 1 : tf3
                     where tf3 = 2 : add tf2 tf3

->> {tf is only used once and can thus be eliminated}
    0 : 1 : tf2
    where tf2 = 1 : tf3
                where tf3 = 2 : add tf2 tf3
```

Sharing: The Benefit of Lazy Evaluation (3)

```
->> {Finally, we obtain successsively longer pre-
    fixes of the stream of Fibonacci numbers}
    0 : 1 : tf2
    where tf2 = 1 : tf3
                where tf3 = 2 : tf4
                            where tf4 = add tf2 tf3
->> 0 : 1 : tf2
    where tf2 = 1 : tf3
                where tf3 = 2 : tf4
                     where tf4 = 3 : add tf3 tf4
    -- Note: eliminating where-clauses corresponds
    -- to garbage collection of unused memory by an
    -- implementation.
->> 0 : 1 : 1 : tf3
                where tf3 = 2 : tf4
                     where tf4 = 3 : add tf3 tf4
```

Note

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

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

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

Evaluating FibsFn shows

- exponential run-time and storage usage!

Memory leak:

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

Illustration

```
fibsFn ()
 ->> 0 : 1 : add (fibsFn ()) (tail (fibsFn ()))
 ->> 0 : tf
     where
       tf = 1 : add (fibsFn ()) (tail (fibsFn ()))
```

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

```
 ->> 0 : tf
     where tf = 1 : add (fibsFn ()) tf
```

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

Chapter 2.2

Stream Diagrams

Stream Diagrams

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

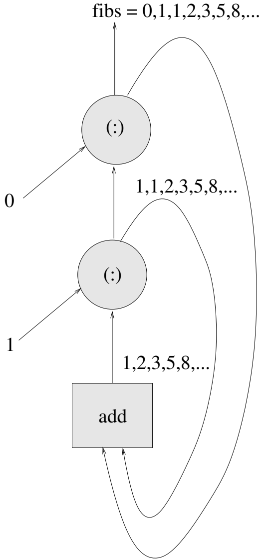
- processes.

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

- Fibonacci numbers
- communications of some client/server application

Example 1: Fibonacci Numbers

...as a stream diagram:



Example 2: A Client/Server Application (1)

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

```
type Request  = Integer
type Response = Integer

client :: [Response] -> [Request]
client ys = 1 : ys   -- issues 1 as the 1st request,
                     -- followed by all responses it
                     -- received (from the server).

server :: [Request]  -> [Response]
server xs = map (+1) xs  -- adds 1 to each request it
                         -- receives (from the client).
```

Two Generators and their Interaction

```
reqs  =  client resps                       -- Generator
resps =  server reqs                        -- Generator
```

Example 2: A Client/Server Application (2)

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

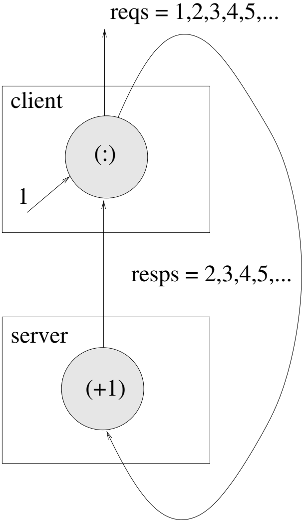
Example 2: A Client/Server Application (3)

Application: Generator/Selector pattern

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

Example 2: The Client/Server Application

...as a stream diagram:



Excursus

Suppose, the client wants to check the first response:

```
client (y:ys) = if ok y then 1 : (y:ys)
                        else error "Faulty Server"
 where ok y = True      -- Trivial check: 'Always
                        -- succeeding'
```

Note: Evaluating

```
reqs ->> client resps
     ->> client (server reqs)
     ->> client (server (client resps))
     ->> client (server (client (server reqs)))
     ->> ...
```

...does not terminate!

The problem: Livelock! Neither the client nor the server can be unfolded! Pattern matching is "too eager."

Remedies: Selector Functions, Lazy Patterns

A): Selector Functions

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

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

B): Lazy patterns (preceding tilde $\sim$)

Defering pattern-matching; no selector function required.

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

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

Illustrating

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

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

reqs ->> client resps
     ->> 1 : if ok y then y:ys
             else error "Faulty Server"
         where y:ys = resps
     ->> 1 : (y:ys)
         where y:ys = resps
     ->> 1 : resps
```

Chapter 2.3

Memoization

Motivation

Memoization is

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

which benefits from

- stream programming.

Memoization

The concept of memoization goes back to Donald Michie:

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

Idea

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

Means

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

Memo Functions, Memo Tables (1)

A memo function is

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

A memo table allows

- to replace recomputation by table look-up.

Soundness of the overall approach:

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

Memo Functions, Memo Tables (2)

Requirement

Let `f : a -> b` be a function. A memo function `memo`

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

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

```
memo f x = f x
```

Making it Concrete: Memo Lists

...as memo tables.

Let `f : Nat -> b` be a (costly to compute) function on natural numbers.

Replace every call of `f` by a look-up in `f_memolist`, which can be considered a (generic) memo list, defined by

```
f_memolist = [ f x | x <- [0..] ]      -- Generator
```

Example 1: Computing Fibonacci Numbers

Computing Fibonacci numbers with memoization/memo lists:

```
fib_memolist = [ fib x | x <- [0..] ]
fib 0  = 0
fib 1  = 1
```

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

Compare this with the naive implementation of `fib`:

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

Lemma 2.3.1

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

Example 2: Computing Powers

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

$$\texttt{pow\_memolist = [power x | x <- [0..]]}$$
$$\texttt{power 0 = 1}$$
$$\texttt{power i = } \underbrace{\texttt{pow\_memolist}}_{\textit{Generator}}\underbrace{\texttt{!!(i-1)}}_{\textit{Selector}} \texttt{ + } \underbrace{\texttt{pow\_memolist}}_{\textit{Generator}}\underbrace{\texttt{!!(i-1)}}_{\textit{Selector}}$$

Compare this with the naive implementation of power:

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

Lemma 2.3.2

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

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

Summing up (1)

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

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

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

Memo functions are

- not a part of the Haskell standard but
- are supported by some non-standard libraries.

Summing up (2)

Important design decision

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

Example:

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

Summing up (3)

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

- Chapter 19, Memoization
  Anthony J. Field, Peter G. Harrison. Functional Programming. Addison-Wesley, 1988.
- Chapter 12.3, Memoization
  Max Hailperin, Barbara Kaiser, Karl Knight. Concrete Abstractions – An Introduction to Computer Science using Scheme. Brooks/Cole Publishing Company, 1999.

Summing up (4)

- Peter J. Landin. *A Correspondence between ALGOL60 and Church's Lambda-Notation: Part I*. Communications of the ACM, 8(2):89-101, 1965.

  ...introduced streams without memoization.

- Daniel P. Friedman, David S. Wise. *CONS should not Evaluate its Arguments*. In Automata, Languages and Programming, 257-281, 1976.

  ...extended Landin's streams with memoization.

- Peter Henderson, James H. Morris. *A Lazy Evaluator*. In Conference Record of the 3rd ACM Symposium on Principles of Programming Languages (POPL'76), ACM, 95-103, 1976.

  ...extended Landin's streams with memoization.

Chapter 2.4

Boosting Performance

Motivation

Recomputating values unnecessarily is a major source of inefficiency:

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

Techniques which can (often) help achieving this are:

- Stream programming
- Memoization

Avoiding Recomputations using Stream Prog.

- Computing Fibonacci numb. using stream programming:

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

  Applications: Generator/Selector pattern

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

- Computing powers using stream programming:

```
powers :: [Integer]                        -- Generator
powers = 1 : 2 : zipWith (+) (tail powers) (tail powers)
```

  Applications: Generator/Selector pattern

```
take 9 powers ->> [1,2,4,8,16,32,64,128,256]
powers!!5 ->> 32
```

- ...

Avoiding Recomputations using Memoization

- Computing Fibonacci numbers using memoization:

```
fib_list = [ fib x | x <- [0..] ]  -- Generator
fib 0 = 0
fib 1 = 1
fib n = fib_list!!(n-1) + fib_list!!(n-2)
```

  Applications: Generator/Selector pattern

```
take 10 fib_list ->> [0,1,1,2,3,5,8,13,21,34]
fiblist!!5 ->> 5
```

- Computing powers using memoization:

```
power_list = [ power x | x <- [0..] ] -- Generator
power 0 = 1
power i = power_list!!(i-1) + power_list!!(i-1)
```

  Applications: Generator/Selector pattern

```
take 9 power_list ->> [1,2,4,8,16,32,64,128,256]
power_list!!5 ->> 32
```

- ...

Summing up

Stream programming and memoization are important though

- no silver bullets

for improving performance by avoiding recomputations.

If, however, they hit they can significantly

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

Obvious candidates

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

Homework: Compare the run-time performance of the straightforward implementations of `fib` and `power` with the one of their "boosted" versions using stream programming and memoization.

Sometimes a Silver Bullet exists

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

Computing Fibonacci numbers provides (again) a striking example.

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

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

Theorem 2.4.1

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

Conclusion

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

- Higher abstraction: Constraining oneself to finite lists is often more complex, and – at the same time – unnatural.
- Modularization: Streams together with lazy evaluation allow for elegant possibilities of decomposing a computational problem. Most important is the
  - **Generator/Prune Paradigm**

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

  and combinations thereof are specific instances of.
- Boosting performance: By avoiding recomputations. Most important are:
  - Stream programming
  - Memoization

Chapter 2.5

References, Further Reading

Chapter 2: Further Reading (1)

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

Chapter 2: Further Reading (2)

- Byron Cook, John Launchbury. *Disposable Memo Functions*. Extended Abstract. In Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97), 310, 1997 (full paper in Proceedings Haskell'97 workshop).
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 7.3, Streams; Chapter 7.8, Memo Functions)
- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 10, Corecursion)

Chapter 2: Further Reading (3)

- Kento Emoto, Sebastian Fischer, Zhenjiang Hu. *Generate, Test, and Aggregate: A Calculation-based Framework for Systematic Parallel Programming with MapReduce*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 254-273, 2012.
- Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988. (Chapter 4.2, Processing 'infinite' data structures; Chapter 4.3, Process networks; Chapter 19, Memoization)
- Daniel P. Friedman, David S. Wise. *CONS should not Evaluate its Arguments*. In Proceedings of the 3rd International Conference on Automata, Languages and Programming, 257-284, 1976.

Chapter 2: Further Reading (4)

- Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions – An Introduction to Computer Science using Scheme*. Brooks/Cole Publishing Company, 1999. (Chapter 12.3, Memoization; Chapter 12.5, Comparing Memoization and Dynamic Programming)
- Peter Henderson, James H. Morris. *A Lazy Evaluator*. In Conference Record of the 3rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'76), 95-103, 1976.
- Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 14, Programming with Streams; Chapter 14.3, Stream Diagrams; Chapter 14.4, Lazy Patterns; Chapter 14.5, Memoization)

Chapter 2: Further Reading (5)

- John Hughes. *Lazy Memo Functions*. In Proceedings of the IFIP Symposium on Functional Programming Languages and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 129-146, 1985.
- Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 6.2, Principles of Dynamic Programming: Memoization or Iteration over Subproblems)
- Peter J. Landin. *A Correspondence between ALGOL60 and Church's Lambda-Notation: Part I*. Communications of the ACM 8(2):89-101, 1965.

Chapter 2: Further Reading (6)

- Donald Michie. *'Memo' Functions and Machine Learning*. Nature, 218:19-22, 1968.
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 14.2.1, Memoization; Kapitel 15.5, Maps, Funktionen und Memoization)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.1, Process networks)
- Jay M. Spitzen, Karl M. Levitt, Lawrence Robinson. *An Example of Hierarchical Design and Proof*. Communications of the ACM 21(12):1064-1075, 1978.

Chapter 2: Further Reading (7)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 17, Lazy programming; Chapter 17.6, Infinite lists; Chapter 17.7, Why infinite lists? Chapter 19.6, Avoiding recomputation: memoization)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17, Lazy programming; Chapter 17.6, Infinite lists; Chapter 17.7, Why infinite lists? Chapter 20.6, Avoiding recomputation: memoization)

Chapter 3

Programming with Higher-Order Functions: Algorithm Patterns

Motivation

Programming with higher-order functions

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

Conceptually

- this emphasises the essence of the underlying algorithmic principle.

Pragmatically

- this makes these algorithmic principles easily re-usable.

Outline

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

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

Chapter 3.1

Divide-and-Conquer

Divide and Conquer

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea:

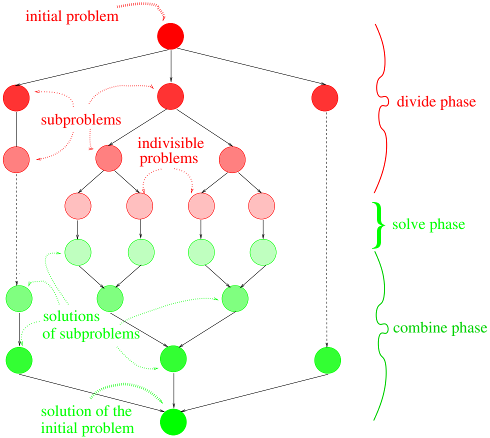
- If a problem instance is simple/small enough, solve it directly or by means of some basic algorithm.
- Otherwise, divide the problem instance into smaller subproblem instances by applying the division strategy recursively until all subproblem instances are simple enough to be solved directly.
- Combine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

Applicability Requirement:

- No generation of identical subproblem instances during problem division.

Illustrating the Divide-and-Conquer Principle

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



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 156.

Implementing Divide-and-Conquer as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Divide-and-Conquer as HoF (2)

The arguments of `divide_and_conquer`:

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

Implementing Divide-and-Conquer as HoF (3)

The HoF Implementation:

$$\texttt{divide\_and\_conquer ::}\ \underbrace{\texttt{(p -> Bool)}}_{\textit{Simple enough?}}\ \texttt{->}\ \underbrace{\texttt{(p -> s)}}_{\textit{Solve!}}\ \texttt{->}$$

$$\underbrace{\texttt{(p -> [p])}}_{\textit{Divide}}\ \texttt{->}\ \underbrace{\texttt{(p -> [s] -> s)}}_{\textit{Combine}}\ \texttt{->}$$

$$\underbrace{\texttt{p}}_{\textit{Problem instance}}\ \texttt{->}\ \underbrace{\texttt{s}}_{\textit{Solution}}$$

```
divide_and_conquer indiv solve divide combine pbi
 = dac pbi
   where
    dac pbi′
     | indiv pbi′ = solve pbi′
     | otherwise  = combine pbi′ (map dac (divide pbi′))
```

Typical Applications of Divide-and-Conquer

Application fields such as

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

Especially

- Quicksort
- Mergesort
- Binomial coefficients
- ...

Example 1: Quicksort

```
quickSort :: Ord a => [a] -> [a]
quickSort lst
 = divide_and_conquer 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
```

Example 2: Fibonacci Numbers (Pitfall!)

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

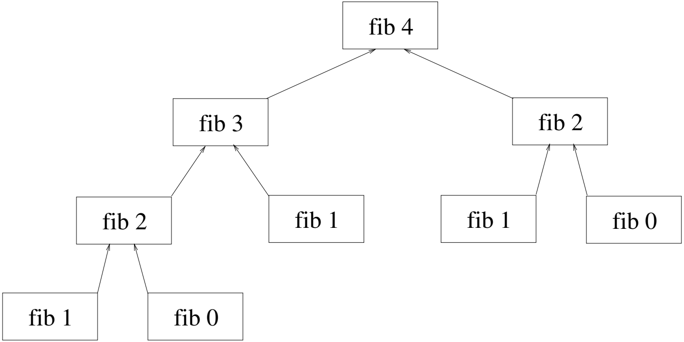
Consider:

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

...shows exponential runtime behaviour due to recomputations!

Illustrating

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



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 179.

Chapter 3.2

Backtracking Search

Backtracking Search

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea:

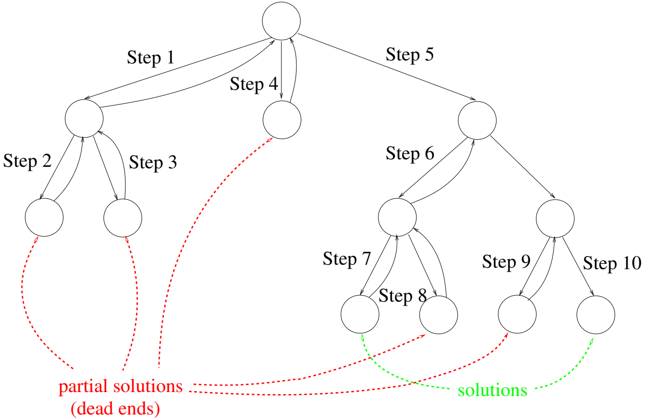
- Search for a particular solution of the problem by a systematic trial-and-error exploration of the solution space.

Applicability Requirements:

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

Illustrating the Backtracking Search Principle

...general stages of a backtracking algorithm:



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 162.

Illustrating Backtracking Search (Cont'd)

Underlying assumptions

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

Note

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

Implementing Backtracking Search as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Backtracking Search as HoF (2)

Note

- Often, the search space is large.

In such cases, the graph forming the search space

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

This reqires

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

Implementing Backtracking Search as HoF (3)

Implementation assumptions:

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

The arguments of `search_dfs`:

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

Implementing Backtracking Search as HoF (4)

The HoF Implementation:

`search_dfs :: (Eq node) =>` $\underbrace{\texttt{(node -> [node])}}_{\textit{Computing successors}}$ `->`

$\underbrace{\texttt{(node -> Bool)}}_{\textit{Solution?}}$ `->`

$\underbrace{\texttt{node}}_{\textit{Initial node}}$ `->` $\underbrace{\texttt{[node]}}_{\textit{Solution nodes}}$

```
search_dfs succ goal n                          -- n for node
 = (search (push n emptyS))
   where
    search s                                    -- s for stack
     | is_emptyS s  = []
     | goal (top s) = top s : search (pop s)
     | otherwise
         = let m = top s
           in search (foldr push (pop s) (succ m))
```

Interface and Behaviour Specification

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

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

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

Implementation A

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

```
data Stack a      = Empty | Stk a (Stack a)

emptyS            = Empty

is_emptyS Empty = True
is_emptyS _       = False

push x s          = Stk x s

pop Empty         = error "Stack is empty"
pop (Stk _ s)     = s

top Empty         = error "Stack is empty"
top (Stk x _)     = x
```

Implementation B

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

```
newtype Stack a    = Stk [a]

emptyS             = Stk []

is_emptyS (Stk []) = True
is_emptyS (Stk _)  = False

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

pop (Stk [])       = error "Stack is empty"
pop (Stk (_:xs))   = Stk xs

top (Stk [])       = error "Stack is empty"
top (Stk (x:_))    = x
```

Typical Applications of Backtracking Search

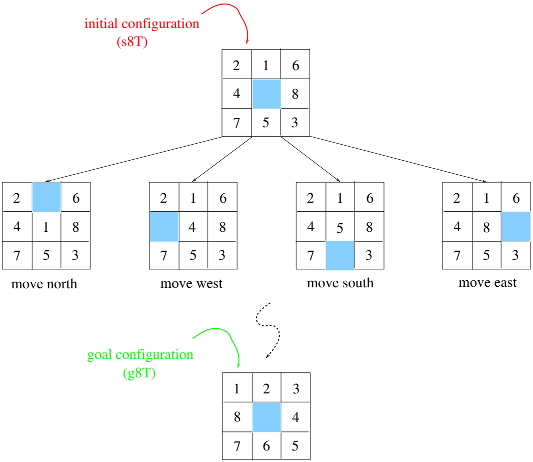
Application fields such as

- Knapsack problems
- Game strategies
- ...

Especially

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

Example: The Eight-Tile Problem (8TP)



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

A Backtracking Search Impl. for 8TP (1)

Modeling the board:

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

The initial board (initial configuration):

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

The final board (goal configuration):

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

A Backtracking Search Impl. for 8TP (2)

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

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

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

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

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

A Backtracking Search Impl. for 8TP (3)

Modeling nodes in the search graph:

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

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

The successor function:

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

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

A Backtracking Search Impl. for 8TP (4)

The goal function:

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

Putting things together:

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

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

Chapter 3.3

Priority-first Search

Priority-first Search (1)

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea

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

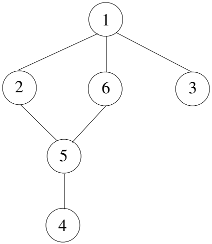
Note: While plain backtracking search proceeds unguidedly and can thus be considered blind, priority-first search/best-first search benefits from (hopefully accurate) information pointing it towards the 'most promising' node.

Priority-first Search (2)

Applicability Requirements

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

Illustrating Different Search Strategies



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 167.

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

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

Implementing Priority-first Search as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Priority-first Search as HoF (2)

Implementation assumptions:

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

The arguments of `search_pfs`:

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

Implementing Priority-first Search as HoF (3)

The HoF Implementation:

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

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

Interface and Behaviour Specification

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

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

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

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

Implementation

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

```
newtype PQueue a   = PQ [a]

emptyPQ            = PQ []

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

enPQ x (PQ pq)     = PQ (insert x pq)
 where
  insert x []                   = [x]
  insert x r@(e:r') | x <= e = x:r -- the smaller the
                                   -- higher the priority
                    | otherwise = e:insert x r'

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

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

Typical Applications of Priority-first Search

Application fields such as

- Game strategies
- ...

Especially

- The eight-tile problem
- ...

Example: A Priority-first Search for 8TP

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

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

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

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

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

Chapter 3.4

Greedy Search

Greedy Search (1)

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea

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

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

Greedy Search (2)

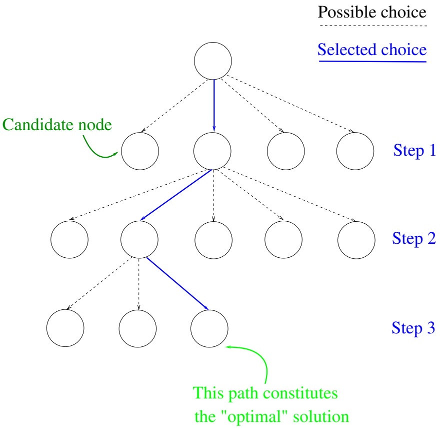
Applicability Requirements

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

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

Illustrating the Greedy Search Principle

...successive stages of a greedy algorithm:



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

Implementing Greedy Search as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Greedy Search as HoF (2)

Implementation assumptions:

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

The arguments of `search_greedy`:

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

Implementing Greedy Search as HoF (3)

The HoF Implementation:

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

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

Note

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

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

Cf. Chapter 3.3 and Chapter 8.4 for details on priority queues as abstract data type (ADT).

Typical Applications of Greedy Search

Application fields such as

- Graph algorithms
- ...

Especially

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

Example: A Greedy Search for MCP (1)

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

Modeling coins:

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

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

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

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

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

Example: A Greedy Search for MCP (2)

The goal function:

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

Putting things together:

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

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

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

Chapter 3.5

Dynamic Programming

Dynamic Programming

Given: A problem instance P.

Sought: A solution S of P.

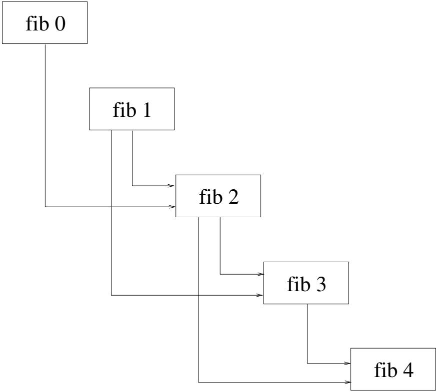
Algorithmic Idea

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

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

Illustrating Dynamic Programming for `fib`

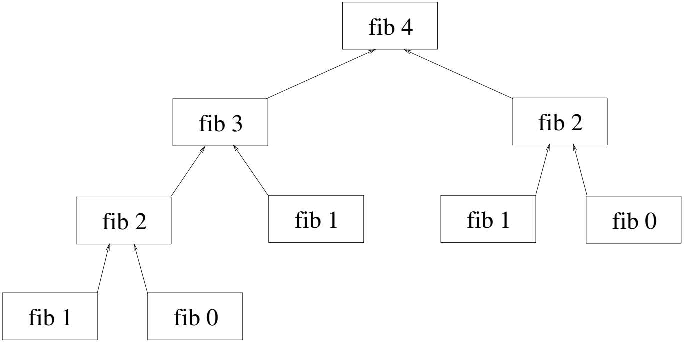
...the dynamic programming computation of the Fibonacci numbers (no recomputation of solutions of subproblems!):



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 179.

Illustrating Divide-and-Conquer for `fib`

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



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 179.

Implementing Dynamic Programming as HoF (1)

Setting:

A problem with

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

Objective:

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

- suitably parameterized problem instances of kind `p` using the 'dynamic programming' principle.

Implementing Dynamic Programming as HoF (2)

The arguments of `dynamic`:

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

Implementing Dynamic Programming as HoF (3)

The HoF Implementation:

```
dynamic :: (Ix coord) =>
              (Table entry coord -> coord -> entry) ->
              Computing the table entry at some coordinates
              (coord,coord)    -> (Table entry coord)
           Specifying table bounds      Result  table

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

Interface/Behaviour Specification

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

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

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

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

Implementation

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

```
newtype Table' a b = Tbl' (Array b a)
new_T' assoc_list = Tbl' (array (low,high) assoc_list)
  where indices = map fst assoc_list
        low     = minimum indices
        high    = maximum indices
find (Tbl' a) index = a ! index
upd_T' p@(index,value) (Tbl' a) = Tbl' (a // [p])
```

Note:

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

Typical Applications of Dynamic Programming

Application fields such as

- Graph algorithms
- Search algorithms
- ...

Especially

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

Example: Computing Fibonacci Numbers

Defining the problem-dependent parameters:

```
bndsFibs :: Int -> (Int,Int)
bndsFibs n = (0,n)

compFib :: Table Int Int -> Int -> Int
compFib t i
 | i <= 1     = i
 | otherwise = find t (i-1) + find t (i-2)
```

Putting things together:

```
fib :: Int -> Int
fib n = find t n
 where t = dynamic compFib (bndsFib n)
```

Dynamic Programming vs. Memoization (1)

Overall

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

Conceptual difference

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

Dynamic Programming vs. Memoization (2)

Minor benefits of dynamic programming

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

Dynamic Programming vs. Memoization (3)

Minor benefits of memoization

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

Chapter 3.6

References, Further Reading

Chapter 3.1–3.4: Further Reading (1)

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

Chapter 3.1–3.4: Further Reading (2)

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

Chapter 3.1–3.4: Further Reading (3)

- Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002. (Kapitel 5, Rekursion - Teile und Herrsche; Kapitel 44, Erschöpfendes Durchsuchen - Backtracking)
- Steven S. Skiena. *The Algorithm Design Manual*. Springer-V, 1998. (Chapter 3.6, Divide and Conquer)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 19.6, Avoiding recomputation: memoization – Greedy algorithms)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 20.6, Avoiding recomputation: memoization – dynamic programming)

Chapter 3.5: Further Reading (4)

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

Chapter 3.5: Further Reading (5)

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

Chapter 3.5: Further Reading (6)

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

Chapter 3.5: Further Reading (7)

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

Chapter 4

Equational Reasoning

Chapter 4.1

Motivation

Functional vs. Imperative Programming (1)

In functional programming

- = means 'equal by definition:' The value of the left-hand side expression is defined as the value of the right-hand side expression.
- Functional definitions of the form

```
f x y = ...
```

  in the definition of a function `f` are thus genuine mathematical equations. The expressions on the left hand side and the right hand side of = have the same value.

Functional vs. Imperative Programming (2)

In imperative programming

- = means 'equality by assignment:' The contents of the memory cell denoted by the left-hand side variable is replaced by the value of the right-hand side expression.
- A symbol sequence of the form

  ```
  x = x+y
  ```

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

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

Functional vs. Imperative Programming (3)

Example: Consider the definition-like symbol sequence S:

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

In functional languages like Haskell, S is an

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

In imperative languages like C or Java, S is a

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

Functional vs. Imperative Programming (4)

Summarizing:

For functional definitions

- standard (algebraic) reasoning about mathematical equations applies.

For imperative assignments

- it does not.

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

Illustrating Equational Reasoning

...on expressions.

Proposition 4.1.1

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

Proof: By equational reasoning we obtain:

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

□

Illustrating Equational Reasoning

...on functional definitions.

Corollary 4.1.2

The Haskell functions f and g defined by

```
f :: Int -> Int -> Int
f a b = (a+b) * (a-b)
g :: Int -> Int -> Int
g a b = a^2 - b^2
```

denote the same function.

Proof: By equational reasoning and Proposition 4.1.1 we obtain:

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

□

More Examples on Equational Reasoning (1)

Let

```
a = 3
b = 4

h :: Int -> Int -> Int
h x y = x^2 + y^2
```

Proposition 4.1.3

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

More Examples on Equational Reasoning (2)

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

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

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

More Examples on Equational Reasoning (3)

Let

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

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

Proposition 4.1.4

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

More Examples on Equational Reasoning (4)

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

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

Folding, Unfolding of Functional Definitions

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

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

in equational reasoning.

Note

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

Let

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

While the first equation `isZero 0 = True`

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

the second equation `isZero n = False`

- can not, since Haskell implicitly imposes an ordering on the equations: Applying the second equation is only legal, if `n` is different from `0`.

Equational Reasoning for Optimization (1)

Note, the straightforward implementation of reverse

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

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

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

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

Equational Reasoning for Optimization (2)

Replacing `reverse` by `fast_reverse` would yield a significant speed-up of programs, provided that `reverse` and `fast_reverse` denote actually the same function.

Using equational reasoning we can in fact prove the equality of `reverse` and `fast_reverse`, and hence justify the above sketched optimation:

Theorem 4.1.5

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

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

Equational Reasoning for Optimization (3)

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

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

$$
\begin{array}{rcl}
 & & \texttt{reverse ls} \\
(\texttt{ls = []}) & = & \texttt{reverse []} \\
(\text{Unfolding } \texttt{reverse}) & = & \texttt{[]} \\
(\text{Folding } \texttt{fr}) & = & \texttt{fr [] []} \\
(\text{Folding } \texttt{fast\_reverse}) & = & \texttt{fast\_reverse []} \\
(\texttt{[] = ls}) & = & \texttt{fast\_reverse ls}
\end{array}
$$

Equational Reasoning for Optimization (4)

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

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

Equational Reasoning for Optimization (5)

Lemma 4.1.6

$\forall$ `ls1,ls2` $\in$ `a-List` $\forall$ `v` $\in$ `a-Value.`

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

Corollary 4.1.7

$\forall$ `ls'` $\in$ `a-List` $\forall$ `v` $\in$ `a-Value.`

`(fr ls' []) ++ [v] = fr ls' [v]`

Proof. Let `ls'` $\in$ `a-List` and let `v` $\in$ `a-Value`. Setting `ls1 = ls'` and `ls2 = []`, Lemma 4.1.6 yields:

		`(fr ls' []) ++ [v]`
`(ls'=ls1, []=ls2)`	=	`(fr ls1 ls2) ++ [v]`
(Lemma 4.1.6)	=	`fr ls1 (ls2 ++ [v])`
`(ls1=ls', ls2=[])`	=	`fr ls' ([] ++ [v])`
`([]++[v]=[v])`	=	`fr ls' [v]`

□

Equational Reasoning for Optimization (6)

Proof of Lemma 4.1.6 by structural induction on the structure of the list argument `ls1` and equational reasoning.

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

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

Equational Reasoning for Optimization (7)

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

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

Equational Reasoning for Optimization (8)

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

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

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

Thus, we have:

Corollary 4.1.8

```
reverse = fast_reverse
```

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

Corollary 4.1.9 (Optimization)

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

Conclusion

Functional definitions are

- genuine mathematical equations.

allowing us to prove

- equality and other relations among functional expressions

by means of usual mathematical reasoning.

Proven equality of functions can justify the replacement of a

- less efficient (called specification) by a more efficient (called implementation) definition of some functionality.

  Two examples:

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

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

Chapter 4.2

Functional Pearls

Functional Pearls: The Very Idea (1)

The design of functional pearls, i.e., functional programs

- evolves from calculation!

In more detail:

Starting from a problem with a

- simple, intuitive but often inefficient specification

we shall arrive at an

- efficient though often more complex and possibly less intuitive implementation

by means of

- mathematical reasoning, i.e., by equational and inductive reasoning, by theorems and laws.

  Example: Transforming `reverse` step by step into `fast_reverse`.

Functional Pearls: The Very Idea (2)

Note: The functional pearl

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

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 (till ca. 2011), some

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

  and other problems.

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

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

Functional Pearls: Origin and Background (3)

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

In his 2011 monograph

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

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

In this chapter

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

- obviously correct but (hopelessly) inefficient

into programs which are

- much more efficient (though possibly less intuitive)

correct:

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

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

Note

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

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

Chapter 4.3

The Smallest Free Number

The Smallest Free Number (SFN) Problem

The SFN-Problem:

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

Examples:

The smallest free number for

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

Analyzing the Problem

Obviously

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

Example:

Computing the smallest free number for the set X

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

SFA: A Straightforward SNFP-Algorithm

The preceding observation suggests the algorithm SFA (reminding to 'StraightForward Algorithm'), which solves the SFN-problem straightforwardly:

The SFNP-Algorithm SFA:

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

Implementing the SFNP-Algorithm SFA

...by means of a system of two functions

- ssfn (reminding to 'simple sfn') and
- sap (reminding to 'search and pick')

Implementation of the SFNP-Algorithm SFA:

```
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 SFN-Problem as a Functional Pearl

The SFNP-Algorithm SFA is sound but inefficient:

- Sorting is not of linear time complexity.

The Functional Pearl View of the SFN-Problem:

Develop an SFNP-Algorithm LinSFNP which is of

- linear time complexity, i.e., which is linear in the number of the elements of the inital set X of natural numbers.

Towards the Linear Time Algorithm

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

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

Here

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

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

```
type Nat = Int
```

is considered the type of natural numbers starting from 0.

Analysing the SFNP-Algorithm SFA′

...by investigating the function `minfree`.

Obviously

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

Note: Evaluating

- `minfree [n-1,n-2..0]`

requires evaluating

- i is not an element in $[n-1, n-2 .. 0]$ for $0 \leq i \leq n$

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

Outline

Starting from SFA′ and `minfree` we will develop

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

linear time algorithms for the SFN-Problem.

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

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

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

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

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

The Array-Based SFNP-Algorithm (1)

Based on KF, the array-based SFNP-Algorithm LinSFNP builds a

- checklist of those numbers present in `filter (<=n) xs`

where checklist is implemented as a

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

Algorithmic idea:

- For each element `x` in `xs` with `x <= 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 SFNP-Algorithm (2)

Implementation of the array-based SFNP-Algorithm LinSFNP:

```
minfree = search . checklist

search :: Array Int Bool -> Int
search = length . takeWhile id . elems

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

Note: The array-based SFNP-Algorithm LinSFNP

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

Variant A of the Array-Based Algorithm

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

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

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

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

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

- boils down to finding the first 0 entry.

Variant B of the Array-Based Algorithm

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

- using a constant-time array update operation.

In Haskell, this can be done using a monad, the

- state monad (cf. `Data.Array.ST`)

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

Note, however, Variant B is essentially a procedural program in functional clothing.

The Divide-and-Conquer SFNP-Algorithm (1)

Algorithmic idea:

- Express `minfree (xs++ys)` in terms of `minfree (xs)` and `minfree (ys)`.

To this end, we first collect some properties of set differences:

Lemma 4.3.1

$$\begin{aligned} (as +\!\!+\ bs) \setminus\!\!\setminus cs &= (as \setminus\!\!\setminus cs) +\!\!+ (bs \setminus\!\!\setminus cs) \\ as \setminus\!\!\setminus (bs +\!\!+\ cs) &= (as \setminus\!\!\setminus bs) \setminus\!\!\setminus cs \\ (as \setminus\!\!\setminus bs) \setminus\!\!\setminus cs &= (as \setminus\!\!\setminus cs) \setminus\!\!\setminus bs \end{aligned}$$

Lemma 4.3.2

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

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

The Divide-and-Conquer SFNP-Algorithm (2)

Lemma 4.3.3

Let `b` be a natural number, and let

- `as = [0..b-1]`, `bs = [b..]`
- `us = filter (<b) xs`, `vs = filter (>=b) xs`

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

Lemma 4.3.3 implies:

Corollary 4.3.4

```
[0..] \ xs = ([0..b-1] \ us) ++ ([b..] \ vs)
                  where (us,vs) = partition (<b) xs
```

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

The Divide-and-Conquer SFNP-Algorithm (3)

Together with

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

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

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

...for any natural number b.

Optimizing DaC-Algorithm LinSFNP′ (1)

Note, evaluating the test

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

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

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

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

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

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

- efficient divide-and-conquer algorithm.

Optimizing DaC-Algorithm LinSFNP′ (2)

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

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

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

Optimizing DaC-Algorithm LinSFNP′ (3)

…provided that `b` is chosen such that both

- `length us` and `length vs` are less than `length xs`

the following recursive definition of `minfree` is well-defined:

```
minfree xs = minfrom 0 xs

minfrom a xs | null xs            = a
             | length us == b-a = minfrom b vs
             | otherwise          = minfrom a us
               where (us,vs) = partition (<b) xs
```

Optimizing DaC-Algorithm LinSFNP′ (4)

...we are left with picking `b` appropriately.

The value of `b` must satisfy:

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

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

```
b = a + 1 + n `div` 2  with  n = length xs.
```

Optimizing DaC-Algorithm LinSFNP′ (5)

Note that

- `n /= 0` and `length us < b-a` implies
  `(length us) <= (n `div` 2) < n`
- `length us = b-a` implies
  `(length vs) = (n - (n `div` 2) - 1) <= n `div` 2`

With this choice, the number of steps for evaluating

```
minfrom 0 xs
```

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

The Optimized DaC-Algorithm LinSFNP″

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

The Optimized Divide&Conquer SFNP-Algorithm LinSFNP″:

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

Conclusion

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

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

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

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

Last but not least

For a 'procedural' programmer

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

For a 'pure functional' programmer

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

This different perception explains why there sometimes

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

Sometimes, however, this gap

- vanishes as for the SFN-Problem.

Chapter 4.4

Not the Maximum Segment Sum

The Maximum Segment Sum (MSS) Problem

A segment of a list

- is a contiguous subsequence.

The MSS-Problem:

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

Example:

Let L be the list

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

The maximum segment sum of L is

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

The MSS-Problem: Background, Motivation

The MSS-Problem

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

In this chapter, however, we consider

- the ‘Maximum Non-Segment Sum (MNSS) Problem’

in the spirit of functional pearl problem.

The Max. Non-Segment Sum (MNSS) Problem

A non-segment of a list

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

The MNSS-Problem:

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

Example:

Let L be the list

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

The maximum non-segment sum of L is

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

What does MNSS qualify a Pearl Problem?

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

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

This means that there are

- many more non-segments of a list than segments

which raises the problem:

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

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

Specifying Solution of the MNSS-Problem

The Specifying Solution of the MNSS-Problem:

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

Intuitively

- 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 Function nonsegs

The implementation of the function nonsegs:

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

relies on the supporting functions

- extract
- nonseg
- markings

which itself relies on the supporting function

- booleans

Implementing the Supporting Functions

...markings, booleans, and extract:

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

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

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

Notes on the Supporting Functions

...`markings`, `booleans`, and `extract`, i.e., the intuition underlying their definitions.

To define the function `nonsegs`

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

This marking

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

Then

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

Notes on the Supporting Function

...`nonseg`:

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

Last but not least:

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

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

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

Note: The regular expression identifies the leftmost gap $T^+ F^+ T$ that makes the segment a non-segment.

The Finite State Automaton

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

```
data State = E | S | M | N
```

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

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

The Implementation of Function nonseg

Implementing nonseg:

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

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

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

Note:

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

Work Plan to Derive the Linear Time Alg.

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

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

Work plan:

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

Towards the Linear Time Algorithm (1)

First, we introduce the function `pick`:

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

Note:

- `nonsegs = pick N`

Towards the Linear Time Algorithm (2)

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

Lemma 4.4.1

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

Towards the Linear Time Algorithm (3)

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

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

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

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

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

Two new Solutions of the MNSS-Problem

The 1st new Solution of the MNSS-Problem:

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

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

Using function `tuple`

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

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

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

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

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

The Fusion Law of `foldl`

Lemma 4.4.2 (Fusion Law of `foldl`)

```
f (foldl g a xs) = foldl h b xs
```

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

```
f a = b
f (g x y) = h (f x) y
```

Towards Applying the Fusion Law (1)

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

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

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

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

```
tuple (maximum . map sum) ([[]], [],[], [])
  = (0, -∞, -∞, -∞)
```

...which gives the definition of `b`.

Towards Applying the Fusion Law (2)

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

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

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

Towards Applying the Fusion Law (3)

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

```
    max (map sum (nss ++ 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 Applying the Fusion Law (4)

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

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

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

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

The Linear Time Algorithm

We are left with dealing with the fictitious ∞ values.

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

The Linear Time Algorithm for the MNSS-Problem:

```
mnss xs
 = fourth (foldl h (start (take 3 xs)) (drop 3 xs))
start [x,y,z]
 = (0, max [x+y+z,y+z,z], max [x,x+y,y], x+z)
```

Conclusions (1)

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

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

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

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

Conclusions (2)

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

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

Chapter 4.5

A Simple Sudoku Solver

Sudoku Puzzles

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

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

The Independent Newspaper

Preliminaries

Preliminary definitions:

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

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

- Grid: A 9×9-matrix of digits.

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

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

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

Assumptions on the Setting

We assume that the input grid is valid, i.e.,

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

Towards the Specifying Solution

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

1st Approach:

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

2nd Approach:

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

In the following we take the 2nd approach to define the specifying initial solution of the Sudoku-problem.

The Specifying Sudoku-Solution

The Specifying Solution of the Sudoku-Problem:

```
solve = filter valid . expand . choices

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

Intuitively:

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

Completing the Specifying Sudoku-Solution (1)

First, we introduce the data type

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

for representing the set of choices.

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

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

Completing the Specifying Sudoku-Solution (2)

Implementing `choices`:

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

Intuitively

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

Completing the Specifying Sudoku-Solution (3)

Implementing expand:

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

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

Intuitively

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

Completing the Specifying Sudoku-Solution (4)

Implementing `valid`:

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

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

Intuitively

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

Completing the Specifying Sudoku-Solution (5)

Implementing `rows` and `columns`:

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

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

Intuitively

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

Completing the Specifying Sudoku-Solution (6)

Implementing `boxs`:

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

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

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

Intuitively

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

Completing the Specifying Sudoku-Solution (7)

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

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

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

Wholemeal Programming

Instead of

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

the preceding approach has gone for functions which

- treat a matrix as a complete entity in itself.

Geraint Jones coined the notion

- wholemeal programming

for this style of programming.

Wholemeal programming

- helps avoiding indexitis and
- encourages lawful program construction.

Lawful Programming

Lemma 4.5.1

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

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

This means, all 3 functions are involutions.

Lemma 4.5.2

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

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

A Quick Analysis of the Specifying Solution

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

Then there are about 9^{40}, or

147.808.829.414.345.923.316.083.210.206.383.297.601

grids to be constructed and checked for validity!

This is hopeless!

Optimizing the Specifying Algorithm

1st Optimization: Pruning the matrix of choices:

Idea

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

Hence, we are seeking for a function

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

which satisfies

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

and realizes the above idea.

Pruning a Row

Pruning a row

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

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

Intuitively

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

Laws for pruneRow, nodeups, and cp

- The function pruneRow satisfies law (G):

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

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

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

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

Rewriting `filter valid . expand`

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

We can prove:

Lemma 4.5.3

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

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

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

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

Proof Sketch of Lemma 4.5.3: `boxs` Case (1)

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

Proof Sketch of Lemma 4.5.3: boxs Case (2)

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

Summing up

Overall, we have shown:

Lemma 4.5.4

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

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

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

Lemma 4.5.5

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

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

Implementation of `solve` after the 1st Opt.

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

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

Note: Pruning can be done more than once.

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

Tuning the Solver Further

...based on the following idea:

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

Which cell to expand?

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

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

Empowering the Function expand

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

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

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

Defining expand1

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

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

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

Remarks on `expand1`

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

Completeness and Safety of a Matrix

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

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

This suggests: A matrix is

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

Note:

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

Testing Completeness and Safety

Completeness and safety can be tested as follows:

- Completeness Test:

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

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

- Safety Test:

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

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

Equational Reasoning

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

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

Implementation of `solve` after the 2nd Opt.

Defining `search` by

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

we have for safe but incomplete matrices the equality

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

This leads us to the final

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

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

Quality and Performance Assessment

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

- `haskell.org/haskellwiki/Sudoku`

It is reported that the solver

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

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

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

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

- equational reasoning!

Chapter 4.6

References, Further Reading

Chapter 4: Further Reading (1)

- Jon R. Bentley. *Programming Pearls*. Addison-Wesley, 1987.
- Jon R. Bentley. *Programming Pearls*. Addison-Wesley, 2nd edition, 2000. (Excerpt of the book online available from `www.cs.bell-labs.com/cm/cs/pearls`)
- Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
- Richard Bird. *Fifteen Years of Functional Pearls*. In Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 215, 2006.

Chapter 4: Further Reading (2)

- Richard Bird. *How to Write a Functional Pearl*. Invited presentation at the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 2006. http://icfp06.cs.uchicago.edu/bird-talk.pdf
- Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011. (Chapter 1, The smallest free number; Chapter 11, Not the maximum segment sum; Chapter 19, A simple Sudoku solver)
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 5, A simple Sudoku solver; Chapter 6.6, The maximum segment sum)
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 4.3.1, Texts as lines)

Chapter 4: Further Reading (3)

- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 9, Formale Überlegungen)
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 10, Applicative Program Transformations)
- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 1.9, Haskell Equations and Equational Reasoning)

Chapter 4: Further Reading (4)

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

Chapter 4: Further Reading (5)

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

Part III

Quality Assurance

Chapter 5

Testing

Chapter 5.1

Motivation

Confidence in Correctness

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

- ours and
- other people's programs

are correct?

Means for Gaining Confidence in Correctness

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

Correctness by Construction (*a priori*, Chapter 4)

- Exemplified by the development of functional pearls.

Verification (*a posteriori*, Chapter 6)

- Rigoros, formal correctness proofs (soundness of the specification, soundness of the implementation).
- High confidence, high effort (typically).

Testing (*a posteriori*, Chapter 5)

- *Ad hoc*: Controllable effort but usually no quantifiable quality statement; hence, a questionable overall value.
- Systematically: Controllable effort, quantifiable quality statement.

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

> Testing can only show the presence of errors.
> Not their absence.
>
> Edsger W. Dijkstra (11.5.1930-6.8.2002)
> *1972 Recipient of the ACM Turing Award*

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

Minimum Requirements for Systematic Testing

Systematic testing should be

- Specification-based
- Tool-supported
- Automatic

Additional 'nice-to-have' features:

Reporting:

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

Reproducibility, Repeatability

- Reproducibility of tests
- Repeatability of tests/testing after program modifications

Specifications

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

- Informally, e.g., as commentary in the program, in a separate documentation
  $\rightsquigarrow$ Disadvantage: often ambiguous, open to interpretation
- Formally, e.g., in terms of pre- and post-conditions, in a formal specification language with a precise semantics
  $\rightsquigarrow$ Advantage: precise and rigorous, unambiguous

In this chapter, we consider systematic testing

...using QuickCheck, a combinator library, which supports specification-based, tool-supported testing in Haskell.

QuickCheck

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

Note

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

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

Chapter 5.2

Defining Properties

Defining Simple Properties w/ QuickCheck (1)

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

Example:

Define inside of a Haskell program the property:

```
prop_PlusAssociative :: Int -> Int -> Int -> Bool
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
```

Defining Simple Properties w/ QuickCheck (2)

...varying the introductory example slightly.

Replace property definition prop_PlusAssociative by:

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

Double-checking the property with Hugs might yield:

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

Note

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

A more Advanced Example

...illustrating limitations of simple property definitions.

Given:

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

To be tested:

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

Property definition:

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

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

Advanced Features for Property Definitions (1)

...using new syntactic features for property definitions:

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

Note:

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

Advanced Features for Property Definitions (2)

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

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

generates randomly a set of sorted lists
tested to satisfy: `is_ordered (insert x xs)`

Note:

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

A Quick Reminder to the Operator ($)

...being defined in the Standard Prelude of Haskell:

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

The ($)-operator is Haskell's infix function application, and useful for saving parentheses:

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

Looking ahead

...QuickCheck supports the specification of more sophisticated properties like e.g.

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

as well as the testing of

- more than one property at the same time.

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

- `Main>quickCheck Module.hs`

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

Chapter 5.3

Testing against Abstract Models

Objective

Testing the correctness (or soundness) of an

- implementation

against a

- reference implementation

of a so-called

- abstract model (or reference model).

We demonstrate this considering an extended example:

- Testing soundness of an efficient implementation of queues against the reference implementation of an abstract model of queues.

The Abstract Model of Queues

...defined in terms of an:

(Executable) Specification:

```
type Queue a = [a]

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

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

Implementing Queues more Efficiently

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

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

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

This queue representation ensures:

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

Example:

- Queue representations: `[7,2,9,4,1,6,8,3]` $\widehat{=}$ `([7,2,9,4],[3,8,6,1])`, `([7,2],[3,8,6,1,4,9])`, `([7,2,9,4,1],[3,8,6])`,...
- Abstract model enqueuing, $(++)$: `[7,2,9,4,1,6,8,3]++[5]`
- Implementation enqueuing, $(:)$: `([7,2,9,4],5:[3,8,6,1])`, `([7,2],5:[3,8,6,1,4,9])`, `([7,2,9,4,1],5:[3,8,6])`,...

Implementing the Abstract Model of Queues

Implementation:

```
type QueueI a      = ([a],[a])

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

is_emptyQI (f,b) = null f
frontQI (x:f,b)  = x

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

Relating Implementation and Abstract Model

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

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

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

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

Example:

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

In the following

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

For convenience, we will focus on queues of integer values (i.e., `(QueueI Int)` and `(Queue Int)`) allowing us to omit

- type specifications in property definitions.

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

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

Soundness Properties: Initial Definitions

Defining five soundness properties:

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

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

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

Falsifiability of prop_isemptyQ

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

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

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

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

  $$\texttt{is\_emptyQI (f,b)} \Rightarrow \texttt{null b}$$

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

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

Falsifiability of `frontQI` and `deQI`

...the definitions of `is_emptyQI`, `frontQI`, and `deQI` all rely on the very same assumption that the front of a queue will only be empty if the back also is.

Therefore, in addition to `prop_isemptyQ` the properties

- `prop_frontQ`
- `prop_deQ`

are falsifiable, too!

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

Soundness Properties: 1st Refinement (1)

We define the invariant as follows:

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

...and adjust the property definitions accordingly:

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

Soundness Properties: 1st Refinement (2)

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

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

However, testing `prop_frontQ` still fails:

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

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

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

Soundness Properties: 2nd Refinement

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

This leads to:

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

Soundness Issues Reconsidered

Now, all five properties (2nd refinement!) pass the Quick-Check test successfully!

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

- operations on queues behave correctly on queues which satisfy the invariant

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

Additionally, we need to check that

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

Additional Soundness Properties

...defining soundness properties for operations producing queues:

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

Testing the Additional Soundness Properties

Testing the additional properties with `QuickCheck` yields:

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

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

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

This means:

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

Fixing the faulty Implementation of enQI

...by replacing the faulty implementation of enQI

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

by the sound one:

```
enQI x (f,b) = flipQ (f,x:b)
 where
  flipQI ([],b) = (reverse b,[])
  flipQI q      = q
```

Now, all 8 properties pass the QuickCheck test successfully!

Summary

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

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

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

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

Chapter 5.4

Testing against Algebraic Specifications

Objective

Testing the correctness (or soundness) of an

- implementation

against

- equational constraints

the operations ought to satisfy, a so-called

- algebraic specification.

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

Algebraic Specification of Queue Operations

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

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

prop_front_emptyQ x = frontQI (enQI x emptyQI) == x
prop_front_enQ x q =
 invariant q && not (is_emptyQI q) ==>
                frontQI (enQI x q) == frontQI q

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

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

Testing against the Algebraic Specification

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

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

Cause of failure: Evaluating

- the left hand side expression yields:
  `deQI (enQI 0 ([1],[0])) ->> deQI ([1],[0,0]) ->> flipQI ([],[0,0]) ->> ([0,0],[])`
- the right hand side expression yields:
  `enQI 0 (deQI ([1],[0])) ->> enQI 0 (flipQI ([],[0])) ->> enQI 0 ([0],[]) ->> ([0],[0])`
- `([0,0],[])` and `([0],[0])` are equivalent (they represent the abstract queue `[0,0]`) but are not exactly equal!

Refining the Algebraic Specification

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

```
q `equiv` q′ = invariant q && invariant q′ &&
                   retrieve q == retrieve q′
```

Replacing the initial formulation of

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

by the new one

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

the QuickCheck test of prop_deQ_enQ passes successfully!

Testing further Equational Constraints

Analogously to the testing approach in Chapter 5.3, we also need to check that

- operations producing a queue do only produce queues which are equivalent, if the arguments are.

To this end, we need to introduce additional soundness properties for the operations enQI and deQI:

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

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

Note

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

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

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

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

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

Looking ahead

...`QuickCheck` provides features to cope with such problems of test case generation; providing especially support for

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

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

Chapter 5.5

Controlling Test Data Generation

Motivation

...by default, the parameters of `QuickCheck` properties are quantified over all values of the underlying data type (e.g., over all integers, over all lists of integers).

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

`QuickCheck` offers several means for controlling quantification over subsets of values.

Dealing with Subsets of Value Domains (1)

Discussed so far: How to deal with subsets of values using

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

Discussed next: How to deal with subsets of values using

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

Dealing with Subsets of Value Domains (2)

...both means differ in their strengths and limitations for particular tasks when chosen for representing relations of values such as being equivalent. Representing equivalence by a

- Boolean function, it is easy to check whether two values are equivalent, but difficult to generate values which are equivalent.
- Generator, i.e., a function from a value to a set of related (here, equivalent) values, it is easy to generate equivalent values, but difficult to check if two given values are equivalent.

While the usage of Boolean functions for representing subsets of values has been illustrated in Chapter 5.3 and Chapter 5.4, the usage of generators will be discussed next.

Type Constructor `Gen`

...defining generators is eased because `Gen` is a monadic type constructor (cf. Chapter 11); for `Gen` as an instance of type constructor class `Monad` holds:

The generator expression

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

Random Element Generation

...by means of function choose, the most basic function of QuickCheck, which makes a choice:

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

Note:

- Random denotes a type class provided by the library module Random of Haskell; its operations support the generation of pseudo-random numbers.
- choose generates a 'random' element of domain a of the specified range.
- choose (1,n), e.g., represents the set {1,...,n}.

Defining Generators using choose

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

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

Note:

- Given a (QueueI a)-value q, equivQ generates a random queue q′, which contains the same elements as q.
- The number k of elements in the back queue of q′ is chosen properly smaller than the total number of elements of q′ (supposed this total number is different from 0).

Property Definitions with Generators (1)

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

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

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

Note:

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

Property Definitions with Generators (2)

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

```
prop_enQ_equivQ q x = invariant q ==>
 forAll (equivQ q) $ \q' -> enQI x q `equiv` enQI x q'

prop_deQ_equivQ q = invariant q && not (null q) ==>
 forAll (equivQ q) $ \q' -> deQI q `equiv` deQI q'
```

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

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

Type-based Generation of Value Sets

...are possible using the overloaded generator `arbitrary`, e.g., for generating the argument values of properties:

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

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

which is equivalent to the short-hand form:

```
prop_max_le x y = x <= x `max` y
```

Type-based Generation of Subsets of Values

...by means of arbitrary and a subsequent value modification as required:

Example: The set of numerical values $\{ y \mid y \geq x \}$ is generated by the generator atLeast defined by:

```
atLeast x = do diff <- arbitrary
               return (x + abs diff)
```

whose definition is based on the set equality:

$$\{ y \mid y \geq x \} = \{ x + abs\ d \mid d \in \mathbb{Z} \}$$

which holds for numerical values,

Note: The idea underlying the definition of atLeast can be adapted to other types than numerical ones.

Selection of Generators

...is enabled by the generator `oneof` which can be thought of as set union operator.

Example: The generator `orderedLists` (cf. Chapter 5.2) generating sorted lists based on the idea that a sorted list is either empty or the result of attaching a new head element to a sorted list of larger elements:

```
orderedLists = do x <- arbitrary
                  listsFrom x
  where
   listsFrom x
    = oneof [return [],              -- either: empty
             do y <- atLeast x       -- or: a list of elems > x
                liftM (x:) (listsFrom y)] -- extended
                                  -- by x as new head element
```

Note

...the `oneof` generator picks alternatives with

- equal probability.

This impacts the generation of test data often unduly. E.g., the generator `orderedLists` will produce

- the empty list far too often

questioning its adequacy and hence usability as a test data generator for ordered lists.

`QuickCheck` offers thus means supporting a weighted selection of generators.

Weighted Selection of Generators

...using the generator `frequency` weights can be assigned to alternatives impacting the relative likelihood they are selected:

```
frequency :: [(Int,Gen a)] -> Gen a
```

Example:

```
listsFrom x
   = frequency [(1,return []),
                (4,do y <- atLeast x
                      liftM (x:) (listsFrom y))]
```

Note:

- `QuickCheck` generators correspond actually to a probability distribution over a set, rather than just the set itself.
- The assignment of weights above gives the cons case a weight of 4; generated lists will thus have an average length of 4 elements.

Making a Generator Default-Gen. for a Type

...if a non-default generator such as `orderedLists` is used frequently, it is advisable to define a new type for the value type it generates and make this new type an instance of the type class `Arbitrary` as shown below:

```
newtype OrderedList a = OL [a]

instance (Num a, Arbitrary a) =>
                    Arbitrary (OrderedList a) where
  arbitrary = liftM OL orderedLists
```

Example: Redefining `insert` with the new type `OrderedList`

```
insert :: Ord a => a -> OrderedList a
                                -> OrderedList a
```

ensures that arguments generated for `insert` will automatically be ordered.

Controlling the Size of Generated Test Data

...is usually necessary in order to avoid the generation of unreasonably large test cases; QuickCheck provides support for this.

QuickCheck generators are parameterized on an

- integer value size, which is gradually increased during testing (first tests explore small cases, later tests larger and larger ones).

The interpretation of the size parameter is up to the

- implementor of a test case generator (the default generator for lists interpretes size as an upper bound on the length).

Generators depending on size can be defined using function

```
sized :: (Int -> Gen a) -> Gen a
```

Example

...the default generator vector for list values:

```
vector n = sequence [arbitrary | i <- [1..n]]
```

...calling `vector` with argument `length` generates lists of random values of length `length`.

`vector` in concert with function `sized`:

```
sized $ \n -> do length <- choose (0,n)
                 vector length
```

The Function `resize`

...allows to supply an explicit size parameter to a generator:

```
resize :: Int -> Gen a -> Gen a
```

Example: Generating a list of lists while bounding the total number of elements by the size parameter:

```
sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
```

Note: The definition uses the default generator but replaces the size parameter by its square root. The list of lists is generated by the default generator `arbitrary` but with a smaller size parameter.

A Note on Lift Functions

...used throughout Chapter 5.5, which are provided by the library module Monad (cf. Chapter 11):

```
liftM  :: Monad m => (a -> b) -> (m a -> m b)
liftM2 :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3 :: Monad m => (a -> b -> c -> d) ->
                             (m a -> m b -> m c -> m d)
liftM4 :: Monad m => (a -> b -> c -> d -> e) ->
                      (m a -> m b -> m c -> m d -> m e)
liftM5 :: Monad m => (a -> b -> c -> d -> e -> f) ->
               (m a -> m b -> m c -> m d -> m e -> m f)
```

Chapter 5.6

Test Data Generators at Work: An Example

Generators for Built-in and User-defined Types

Test data generators for

- predefined ('built-in') types of Haskell
  - are provided by `QuickCheck`.
- user-defined types
  - must be provided by the user in terms of defining suitable instances of the type class `Arbitrary`.
  - require usually measures to control the size of generated test data, especially for inductively defined types.

...this is illustrated next considering a binary tree type.

A User-defined Generator for Binary Trees (1)

...we consider the following binary tree type:

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

`(Tree a)` can straightforwardly be made an instance of type class Arbitrary:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency [(1,return Leaf),
               (3,liftM3 Branch
                   arbitrary arbitrary arbitrary)]
```

A User-defined Generator for Binary Trees (2)

Note:

- The assignment of weights (1 vs. 3) shall ensure that not too many trivial trees of size 1 are generated.
- Problem: The likelihood that a finite tree is generated, is only one third because termination is only possible, if all subtrees which are generated are finite. With increasing breadth of the generated trees, the requirement of always selecting the 'terminating' branch must be satisfied at ever more places simultaneously...

Remedy: Using the `size` parameter in order to ensure

- termination and
- generation of trees of 'reasonable' size.

A User-defined Generator for Binary Trees (3)

...replace the initial `instance`-declaration for `(Tree a)` by:

```
instance Arbitrary a => Arbitrary (Tree a) where
 arbitrary = sized arbTree

arbTree 0 = return Leaf
arbTree n | n>0 =
 frequency [(1,return Leaf),
            (3,liftM3 Branch shrub arbitrary shrub)]
  where shrub = arbTree (n `div` 2)
```

Note:

- `shrub` is a generator for 'small(er)' trees. It is not bound to a special tree; the two occurrences of `shrub` will usually generate different trees.
- Since the size limit for subtrees is halved, the total size is bounded by the parameter `size`.
- Generators for recursive types must usually be handled like in this example.

Chapter 5.7

Monitoring, Reporting, and Coverage

Test-Data Monitoring

In practice, it is useful to monitor the

- generated test cases in order to obtain a hint on the quality and the coverage of test cases

of a `QuickCheck` run.

For this purpose `QuickCheck` provides a bunch of

- monitoring and reporting possibilities.

Why is Test-Data Monitoring Required?

...reconsider the example of inserting into a sorted list:

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

`QuickCheck` checks `prop_InsertOrdered` by

- randomly generating lists and checking every one, whether it is sorted (used as test case) or not (discarded).

Obviously, the likelihood that a randomly generated list

- is sorted is the higher the shorter the list is.

This introduces the danger that

- property `prop_InsertOrdered` is mostly tested with lists of length one or two.
- even a successful test is not meaningful.

QuickCheck Combinators

...allowing to control test-data monitoring:

- `trivial`
- `classify`
- `collect`

Test-Data Monitoring using `trivial` (1)

The combinator `trivial` is useful for monitoring purposes, where

- the meaning of 'trivial' is user-definable, e.g., that lists up to a length of 2 are considered trivial.

Example:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs = is_ordered xs ==>
 trivial (length xs <= 2) $ is_ordered (insert x xs)
```

Double-checking the property with Hugs yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial).
```

Test-Data Monitoring using `trivial` (2)

Observation regarding the example:

- 91% are too many trivial test cases in order to ensure that the total test is meaningful.
- The operator `==>` should thus be used with care in test-case generators.

Remedy:

- User-defined generators, e.g., by using quantification as sketched in Chapter 5.2.

Note:

- The combinator `trivial` is defined in terms of the more general combinator `classify`:

```
trivial p = classify p "trivial"
```

Test-Data Monitoring using `classify`

The combinator `classify` allows a more refined test-case monitoring than the combinator `trivial`.

Example:

```
prop_InsertOrdered x xs = is_ordered xs ==>
 classify (null xs) "empty lists" $
  classify (length xs == 1) "unit lists" $
   is_ordered (insert x xs)
```

Double-checking this property yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```

Test-Data Monitoring using collect

Going beyond, the combinator collect allows to keep track on all test cases.

Example:

```
prop_InsertOrdered x xs = is_ordered xs ==>
  collect (length xs) $ is_ordered (insert x xs)
```

Double-checking this property yields a histogram of values:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
 5% 3.
```

Chapter 5.8

Implementation of QuickCheck

QuickCheck: Facts and Figures

QuickCheck

- consists in total of about 300 lines of code.
- has been developed by Koen Claessen and John Hughes and initially presented in:

  Koen Claessen, John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.

QuickCheck: A Glimpse of the Code

```
newtype Property = Prop (Gen Result)

class Testable a where
 property :: a -> Property

instance Testable Bool where
 property b = Prop (return (resultBool b))

instance Testable Property where
 property p = p

instance (Arbitrary a, Show a, Testable b) =>
                             Testable (a -> b) where
 property f = forAll arbitrary f

quickCheck :: Testable a => a -> IO ()
```

Background Material

For further details, also on applications, refer e.g., to:

- Koen Claessen, John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- Koen Claessen, John Hughes. Testing Monadic Code with QuickCheck. In Proceedings of the ACM SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.

as well as to:

- Koen Claessen, John Hughes. Specification-based Testing with QuickCheck. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 17-39, 2003.

...on which the presentation of this chapter is closely based on.

Chapter 5.9

Summary

Relevance and Value of Specifications

Experience shows:

- Formalizing specifications is meaningful (even without a subsequent formal proof of soundness).
- Specifications provided are (initially) often faulty themselves.

Relevance and Value of Testing

Investigations of Richard Hamlet indicate that

- a high number of test cases yields meaningful results even in the case of random testing.
- the generation of random test cases is often 'cheap.'

Hence, there are many good reasons advising

- the routine usage of tools like `QuickCheck`!

Fur further details, refer to:

- Richard Hamlet. Random Testing. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994.

Relevance and Value of QuickCheck

Experience shows that `QuickCheck` is an effective tool for

- disclosing bugs in programs and specifications with little effort.
- reducing test costs while at the same time testing more thoroughly.

Related Approaches

...besides `QuickCheck` there are various other `combinator libraries` supporting the lightweight testing of Haskell programs, e.g.:

- EasyCheck
- SmallCheck
- Lazy SmallCheck
- Hat (for tracing Haskell programs)

A Confirmation of the Relevance of Testing

...also by:

C. Antony Hoare (* 1934)
Recipient of the 1980 ACM A.M. Turing Award:
For his fundamental contributions to the definition and design of programming languages.

An influential work, advocating rigor and correctness:

- Charles A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10):576-580, 1969.

40 years later, a retrospective:

- Charles A.R. Hoare. Retrospective: An Axiomatic Basis for Computer Programming. Communications of the ACM 52(10):30-32, 2009.

A Quote from Hoare's Retrospective Article

"One thing I got spectacularly wrong. I could see that programs were getting larger, and I thought that testing would be an increasingly ineffective way of removing errors from them. I did not realize that the success of tests is that they test the programmer, not the program. Rigorous testing regimes rapidly persuade error-prone programmers (like me) to remove themselves from the profession. Failure in test immediately punishes any lapse in programming concentration, and (just as important) the failure count enables implementers to resist management pressure for premature delivery of unreliable code [...]. The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct. Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it. My basic mistake was to set up proof in opposition to testing, where in fact both of them are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs."

Chapter 5.10

References, Further Reading

Chapter 5: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18.2, QuickCheck)
- F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.
- Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings of the ACM SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.

Chapter 5: Further Reading (2)

- Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 17-39, 2003.
- Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, Malcolm Wallace. *Testing and Tracing Lazy Functional Programs Using `QuickCheck` and `Hat`*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 59-99, 2003.
- Jan Christiansen, Sebastian Fischer. *Easycheck – Test Data for Free*. In Proceedings of the 9th International Symposium on Functional and Logic Programming (SFLP 2008), Springer-V., LNCS 4989, 322-336, 2008.

Chapter 5: Further Reading (3)

- Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994.
- Charles A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM 12(10):576-580, 1969.
- Charles A.R. Hoare. *Retrospective: An Axiomatic Basis for Computer Programming*. Communications of the ACM 52(10):30-32, 2009.
- Colin Runciman, Matthew Naylor, Fredrik Lindblad. *SmallCheck and Lazy SmallCheck*. In Proceedings of the ACM SIGPLAN 2008 Haskell Workshop (Haskell 2008), 37-48, 2008. (Available from `http://hackage.haskell.org`)

Chapter 5: Further Reading (4)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 11, Testing and Quality Assurance; Chapter 26, Advanced Library Design: Building a Bloom Filter – Testing with QuickCheck)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 19.6, DSLs for computation: generating data in QuickCheck)

This software comes "without warranty of any kind, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose."

Chapter 6

Verification

Motivation

...testing and verification aim both at

- ensuring the correctness of a program or system

but are of different rigor.

Though testing can be amazingly effective, it is limited to

- showing the presence of errors. It can not show their absence (except of the most simple scenarios).

By contrast, verification can

- prove the absence of errors!

In this chapter

...we will consider important inductive proof principles for proving properties of functional programs (though not limited to functional programs) which may operate on

- unstructured data
  - integers
  - chars
  - Booleans
  - ...
- structured data
  - lists (finite by definition)
  - streams (infinite by definition)
  - trees (finite or infinite)
  - ...

Inductive Proof Principles: Outline

...we will consider:

- Inductive proof principles on natural numbers
  - Natural (or mathematical) induction (dtsch. vollständige Induktion)
  - Strong induction (dtsch. verallgemeinerte Induktion)
- Inductive proof principles on structured data
  - Structural induction (dtsch. strukturelle Induktion)

  In particular:
  - Structural induction on lists
  - Structural induction on stream approximants
- Coinduction
- Fixed point induction

Ohne Mathematik tappt man doch immer im Dunkeln.

Werner von Siemens (1816-1892)
dt. Erfinder und Unternehmer

Chapter 6.1

Inductive Proof Principles on Natural Numbers

Chapter 6.1.1

Natural Induction

The Principle of Natural Induction

Let $\mathbb{N}$ be the set of natural numbers, and P be a property of natural numbers.

The Principle of Natural (or Mathematical) Induction

$$\underbrace{P(1)}_{\text{Base Case}} \wedge [\overbrace{\forall n \in \mathbb{N}.\ \underbrace{P(n)}_{\text{Induction Hypothesis}} \Rightarrow \underbrace{P(n+1)}_{\text{Induction Step}}}^{\text{Inductive Case}}] \quad \Rightarrow \quad \underbrace{\forall\, n \in \mathbb{N}.\ P(n)}_{\text{Conclusion}}$$

(dtsch. Prinzip der vollständigen Induktion)

Example: Illustrating Natural Induction

Lemma 6.1.1.1

$$\forall\, n \in \mathbb{N}.\ \sum_{k=1}^{n} (2k-1) = n^2$$

Proof by means of natural (mathematical) induction.

Proof of Lemma 6.1.1.1 (1)

Base case: Let $n = 1$. In this case we obtain the equality of the left and right hand side expression straightforwardly by equational reasoning:

$$\begin{aligned}
\sum_{k=1}^{n}(2k-1) &= \sum_{k=1}^{1}(2k-1) \\
&= 2*1-1 \\
&= 2-1 \\
&= 1 \\
&= 1^2 \\
&= n^2
\end{aligned}$$

Proof of Lemma 6.1.1.1 (2)

Inductive case: Let $n \in \mathbb{N}$. By means of the induction hypothesis (IH) we can assume $\sum_{k=1}^{n}(2k-1) = n^2$. This allows us to complete the proof as follows:

$$\begin{array}{rcl}
\sum_{k=1}^{n+1}(2k-1) & = & 2(n+1)-1+\sum_{k=1}^{n}(2k-1) \\
\text{(IH)} & = & 2(n+1)-1+n^2 \\
 & = & 2n+2-1+n^2 \\
 & = & 2n+1+n^2 \\
 & = & n^2+2n+1 \\
 & = & n^2+n+n+1 \\
 & = & (n+1)(n+1) \\
 & = & (n+1)^2
\end{array}$$

$\square$

Homework

Prove by means of natural (mathematical) induction:

Lemma 6.1.1.2

1. $$\forall\, n \in \mathbb{N}.\ \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

2. $$\forall\, n \in \mathbb{N}.\ \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

3. $$\forall\, n \in \mathbb{N}.\ \sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$$

Chapter 6.1.2

Strong Induction

The Principle of Strong Induction

Let $\mathbb{N}$ be the set of natural numbers, and P be a property of natural numbers.

The Principle of Strong Induction

$$\forall n \in \mathbb{N}.\ [\overbrace{\underbrace{(\forall m < n.\ P(m))}_{\text{Induction Hypothesis}} \Rightarrow \underbrace{P(n)}_{\text{Induction Step}}}^{\text{(Inductive) Case}}] \quad \Rightarrow \quad \underbrace{\forall\, n \in \mathbb{N}.\ P(n)}_{\text{Conclusion}}$$

(dtsch. Prinzip der verallgemeinerten Induktion)

Note: For the smallest natural number $\hat{n}$ ($\mathbb{N}_0$ vs. $\mathbb{N}_1$), the induction hypothesis boils down to 'true', i.e., $P(\hat{n})$ has to be proven without relying on anything special.

Example: Illustrating Strong Induction

The Fibonacci function is defined by:

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

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

Lemma 6.1.2.1

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

Proof by means of strong induction.

The Key for Proving Lemma 6.1.2.1

...for the case $n \in \mathbb{N}_0$, $n \geq 2$, is to use the equality

$$fib(m) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^m - \left(\frac{1-\sqrt{5}}{2}\right)^m}{\sqrt{5}}$$

given by the induction hypothesis (IH) for $m = n - 1$ and $m = n - 2$.

(Note: In the case of $n \geq 2$, we could use this equality even for all $m < n$ by means of the induction hypothesis (instead of only for $m = n - 1$ and $m = n - 2$). This, however, is not required to complete the proof.)

Proof of Lemma 6.1.2.1 (1)

Case 1: Let $n = 0$. Equational reasoning yields straightforwardly the desired equality:

$$fib(0) = 0 = \frac{0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}}$$

(Note: For proving Case 1, the induction hypothesis allows nothing to assume on the validity of the statement. Fortunately, nothing is required.)

Case 2: Let $n = 1$. Again, equational reasoning yields directly the desired equality:

$$fib(1) = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\frac{1}{2} + \frac{\sqrt{5}}{2} - (\frac{1}{2} - \frac{\sqrt{5}}{2})}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}}$$

(Note: For proving Case 2, we could have used the statement for $n = 0$ by means of the induction hypothesis. This, however, is not required.)

Proof of Lemma 6.1.2.1 (2)

Case 3: Let $n \in \mathbb{N}_0$, $n \geq 2$. Using IH for $n-2, n-1$ we obtain as desired:

$$\begin{aligned}
fib(n) &= fib(n-2) + fib(n-1) \\
(2\text{x IH}) &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}} \\
&= \frac{\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\right] - \left[\left(\frac{1-\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1+\sqrt{5}}{2}\right] - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1-\sqrt{5}}{2}\right]}{\sqrt{5}} \\
(*) &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left(\frac{1+\sqrt{5}}{2}\right)^{2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left(\frac{1-\sqrt{5}}{2}\right)^{2}}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}}{\sqrt{5}}
\end{aligned}$$

□

Proof of Equality $(*)$

The equality marked by $(*)$ holds because of the two equalities shown below which are proved by equational reasoning using the binomial formulae:

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = 1+\frac{1+\sqrt{5}}{2}$$

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = 1+\frac{1-\sqrt{5}}{2}$$

Homework

Let function f be defined by:

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

$$f(n) =_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \sum_{k=0}^{n-1} f(k) & \text{if } n \geq 2 \end{cases}$$

Prove by means of (Lemma 6.1.2.3 and) strong induction :

Lemma 6.1.2.2

$$(\forall\, n \in \mathbb{N}.\ n \geq 2).\ f(n) = 2^{n-2}$$

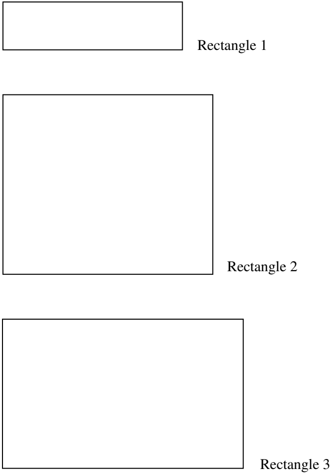
Prove by natural (mathematical) induction:

Lemma 6.1.2.3

$$(\forall\, n \in \mathbb{N}.\ n \geq 3).\ \sum_{k=0}^{n-3} 2^k = 2^{n-2} - 1$$

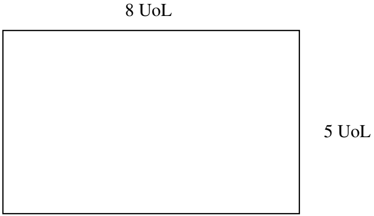
Excursus: Which Rectangle

...is the 'most' typical, the 'nicest' rectangle?



....most people say 'Rectangle 3'!

Why?

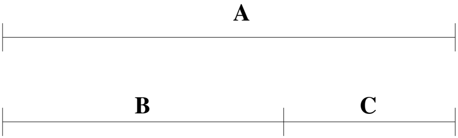


$$8\ UoL/5\ UoL = 1.6 \quad (\text{UoL} \mathrel{\widehat{=}} \text{Unit of Length})$$

The value 1.6 comes close to

...the Golden Ratio:

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803398874989\ldots$$



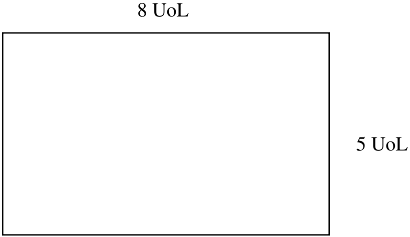
Note: The ratio of section A and section B, denoted by ϕ and called the Golden Ratio, is the same as the ratio of section B and section C:

$$\phi =_{df} A/B = B/C$$

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
6.1
6.1.1
6.1.2
6.2
6.3
6.4
6.5
6.6
6.7
6.8
Part IV
Chap. 7
Chap. 8

The Golden Ratio

...is perceived by most people as very harmonious:



$$8\ UoL / 5\ UoL = 1.6$$

Computing ϕ

$$\begin{array}{rcccc} & \frac{x+1}{x} & =_{df} & \phi & = & \frac{x}{1} \\ \iff & 1+\frac{1}{x} & = & \phi & = & x \end{array}$$

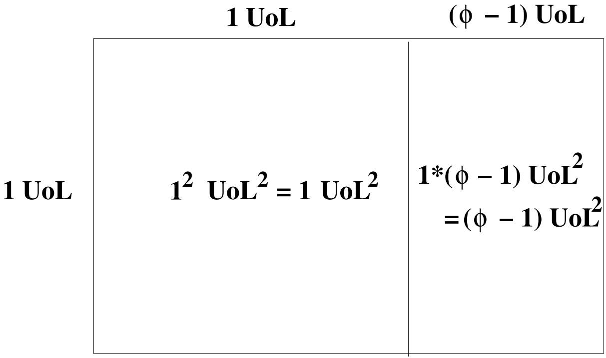
Using ϕ for x, we get:

$$\begin{array}{rlcl} & 1+\frac{1}{\phi} & = & \phi \\ \iff & \phi+1 & = & \phi^2 \\ \iff & 0 & = & \phi^2-\phi-1 \\ \iff & \phi = \frac{1+\sqrt{5}}{2} = 1.618\ldots & & \\ & \phi' = \frac{1-\sqrt{5}}{2} = -0.618\ldots & & \end{array}$$

Note: ϕ' lacks a geometrical interpretation.

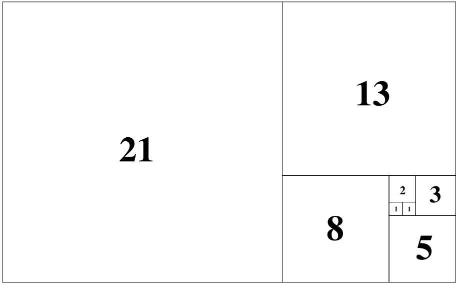
The Golden Ratio

...shows up not only as the ratio of sections but also as the ratio of areas, e.g., rectangles:



The Golden Ratio

...related also to (the ratio of subsequent) Fibonacci numbers:



Illustration

The sequence of Fibonacci numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

The sequence of the ratios of the Fibonacci numbers:

$$\begin{array}{rcl}
1/1 & = & 1 \\
2/1 & = & 2 \\
3/2 & = & 1.5 \\
5/3 & = & 1.\overline{6} \\
8/5 & = & 1.6 \\
13/8 & = & 1.625 \\
21/13 & = & 1.615384615384615 \\
34/21 & = & 1.619047619047619 \\
\ldots & & \ldots \\
1,346,269/832,040 & = & 1.618033988750541 \approx \phi
\end{array}$$

The Golden Ratio

...as the limit of the ratios of the Fibonacci numbers:

$$\lim_{n\to\infty} \frac{fib(n+1)}{fib(n)} = \frac{1+\sqrt{5}}{2} = \phi$$

...letting Lemma 6.1.2.1 perhaps less arbitrarily looking than it might do at first sight.

Chapter 6.2

Inductive Proof Principles on Structured Data

Chapter 6.2.1

Induction and Recursion

Induction and Recursion

...are closely related.

Induction

- describes things starting from something very simple, and building up from there: A bottom-up principle.

Recursion

- starts from the whole thing, working backward to the simple case(s): A top-down principle.

Hence:

- Induction (bottom-up) and recursion (top-down) can be considered the two sides of the same coin.

In fact

...the preferred usage of induction over recursion in some contexts resp. vice versa

- e.g., defining data structures (induction)
- e.g., defining algorithms (recursion)

is often mostly due to historical reasons.

Data type (inductively defined)

```
data Tree = Leaf Int | Node Tree Int Tree
```

Algorithm (recursively defined)

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

Illustration

- Inductive definition of (simple) arithmetic expressions:
  (r1) Each numeral n and variable v is an (elementary) arithmetic expression.
  (r2) If e_1 and e_2 are arithmetic expressions, then also $(e_1 + e_2)$, $(e_1 - e_2)$, $(e_1 * e_2)$, and (e_1/e_2).
  (r3) Every arithmetic expression is inductively constructed by means of rules (r1) and (r2).

- Recursive definition of the merge sort algorithm:
  A list of integers l is sorted by the following 3 steps:
  (ms1) Split l into two sublists l_1 and l_2.
  (ms2) Sort the sublists l_1 and l_2 recursively obtaining the sorted sublists sl_1 and sl_2.
  (ms3) Merge the sorted sublists sl_1 and sl_2 into the sorted list sl of l.

Summary

Data structures often follow an

- inductive definition pattern, e.g.:
  - A list is either empty or a pair consisting of an element and another list.
  - A (binary) tree is either a leaf or consists of a node and a left and a right subtree.
  - An arithmetic expression is either a numeral or a variable, or is composed of (two) arithmetic expressions by means of a (binary) arithmetic operator.

Algorithms (functions) on data structures often follow a

- recursive definition pattern, e.g.:
  - The function `length` computing the length of a list.
  - The function `depth` computing the depth of a tree.
  - The function `evaluate` computing the value of an arithmetic expression (given a valuation of its variables).

Chapter 6.2.2

Structural Induction

The Principle of Structural Induction

Let S be a set of elements inductively constructed from finitely many simpler/simplest elements of S, let $sub(s) \subseteq S$, $s \in S$, denote the finite set of elements s is constructed from, and let P be a property of the elements of S.

The Principle of Structural Induction

$$\forall\, s \in S.\ [\overbrace{\underbrace{(\forall\, s' \in sub(s).\ P(s'))}_{\text{Induction Hypothesis}} \Rightarrow \underbrace{P(s)}_{\text{Induction Step}}}^{\text{(Inductive) Case}}] \quad \Rightarrow \quad \underbrace{\forall\, s \in S.\ P(s)}_{\text{Conclusion}}$$

(dtsch. Prinzip der strukturellen Induktion)

Note: For the 'simplest' elements (or atoms or building blocks) $\hat{s}$ of S we have $sub(\hat{s}) = \emptyset$. For these elements the induction hypothesis boils down to 'true', i.e., $P(\hat{s})$ has to be proven without relying on anything special.

Example: Illustrating Structural Induction

Let the set of (simple) arithmetic expressions $\mathcal{AE}$ be defined by the BNF rule:

$$e \;::=\; n \mid v \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 * e_2) \mid (e_1 / e_2)$$

where n and v stand for (integer) numerals and variables, respectively.

Lemma 6.2.2.1

Let p_e and op_e denote the number of parentheses and operators of any expression e, $e \in \mathcal{AE}$, respectively. Then:

$$\forall\, e \in \mathcal{AE}.\; p_e \;=\; 2 * op_e$$

Proof by means of structural induction.

Proof of Lemma 6.2.2.1 (1)

(Base) case: Let $e \equiv n$, n a numeral, or $e \equiv v$, v a variable.

In both cases e does not contain any parentheses or operators. This means $p_e = 0 = op_e$. This yields directly the desired equality:

$$\begin{aligned} & p_e \\ = \; & 0 \\ = \; & 2 * 0 \\ = \; & 2 * op_e \end{aligned}$$

Proof of Lemma 6.2.2.1 (2)

(Inductive) case: Let $e \equiv (e_1 \circ e_2)$, $\circ \in \{+, -, *, /\}$, and $e_1, e_2 \in \mathcal{AE}$. By means of the induction hypothesis (IH), we can assume $p_{e_1} = 2 * op_{e_1}$ and $p_{e_2} = 2 * op_{e_2}$. The equality of p_e and $2 * op_e$ follows then by equational reasoning:

$$
\begin{array}{rcl}
 & & p_e \\
(\, e \equiv (e_1 \circ e_2)\,) & = & p_{(e_1 \circ e_2)} \\
 & = & 1 + p_{e_1} + p_{e_2} + 1 \\
(\text{2x IH}) & = & 2 * op_{e_1} + 2 + 2 * op_{e_2} \\
 & = & 2 * op_{e_1} + 2 * 1 + 2 * op_{e_2} \\
 & = & 2 * (\, op_{e_1} + 1 + op_{e_2}\,) \\
 & = & 2 * op_{(e_1 \circ e_2)} \\
(\,(e_1 \circ e_2) \equiv e\,) & = & 2 * op_e
\end{array}
$$

$\square$

Homework (1)

Prove by means of structural induction:

Lemma 6.2.2.2

Let lp_e and rp_e denote the number of left and right parentheses of any expression $e \in \mathcal{AE}$, respectively. Then:

$$\forall\, e \in \mathcal{AE}.\ lp_e \;=\; rp_e$$

Lemma 6.2.2.3

Let d_e and opd_e denote the depth and the number of operands of any expression $e \in \mathcal{AE}$, respectively. Then:

$$\forall\, e \in \mathcal{AE}.\ opd_e \leq 2^{d_e}$$

Homework (2)

An arithmetic expression is called

- finite, if the length of all paths originating at its root operator is finite.
- complete, if it is finite and all paths from an operand to the root operator are of the same length.

Lemma 6.2.2.4

Let d_e and opd_e denote the depth and the number of operands of any expression $e \in \mathcal{AE}$, respectively. Then:

$$(\forall\, e \in \mathcal{AE}.\ e\ complete).\ opd_e = 2^{d_e}$$

Proof by means of structural induction.

Note

...the principles of

- natural (math.) induction (dtsch. vollständige Induktion)

  $P(1) \wedge [\forall n \in \mathbb{N}.\, P(n) \Rightarrow P(n+1)] \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\, P(n)$

- strong induction (dtsch. verallgemeinerte Induktion)

  $\forall n \in \mathbb{N}.\, [(\forall m < n.\, P(m)) \Rightarrow P(n)] \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\, P(n)$

- structural induction (dtsch. strukturelle Induktion)

  $\forall\, s \in S.\, [(\forall\, s' \in sub(s).\, P(s')) \Rightarrow P(s)] \;\Rightarrow\; \forall\, s \in S.\, P(s)$

are equally expressive.

Chapter 6.3

Inductive Proofs on Algebraic Data Types

Chapter 6.3.1

Inductive Proofs on Haskell Trees

Inductive Proofs on Finite Trees

A tree is called

- finite, if the length of all paths originating at its root is finite.
- maximum, if it is finite and all paths from a leaf to its root are of the same length.

Let

```
data Tree = Leaf Int | Node Tree Tree
```

Lemma 6.3.1.1

Let *depth*(t) and *leaves*(t) denote the depth and the number of leaves of any finite tree value `t :: Tree`, respectively. Then:

$$(\forall\, \texttt{t :: Tree}.\ \texttt{t}\ \mathit{maximum}).\ \mathit{leaves}(\texttt{t}) = 2^{\mathit{depth}(\texttt{t})}$$

Proof by means of structural induction.

Proof of Lemma 6.3.1.1 (1)

Base case: Let $\texttt{t} \equiv (\texttt{Leaf k})$ for some integer value $\texttt{k}$.

Here, we have $depth(\texttt{t}) = 0$ and $leaves(\texttt{t}) = 1$. Equational reasoning yields the desired equality of $leaves(\texttt{t})$ and $2^{depths(\texttt{t})}$:

$$
\begin{array}{rcl}
 & & leaves(\texttt{t}) \\
(\texttt{t} \equiv (\texttt{Leaf k})) & = & leaves(\texttt{Leaf k}) \\
 & = & 1 \\
 & = & 2^0 \\
 & = & 2^{depths(\texttt{t})}
\end{array}
$$

Proof of Lemma 6.3.1.1 (2)

Inductive case: Let $\texttt{t} \equiv (\texttt{Node t1 t2})$ maximum. This implies $\texttt{t1}$, $\texttt{t2}$ are maximum themselves, $\mathit{depth}(\texttt{t1}) = \mathit{depth}(\texttt{t2})$, and $\mathit{depth}(\texttt{t}) = \mathit{depth}(\texttt{t1}) + 1 = \mathit{depth}(\texttt{t2}) + 1$. By means of the inductive hypothesis (IH) we can assume $\mathit{leaves}(\texttt{t1}) = 2^{\mathit{depth}(\texttt{t1})}$ and $\mathit{leaves}(\texttt{t2}) = 2^{\mathit{depth}(\texttt{t2})}$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \mathit{leaves}(\texttt{t}) \\
(\,\texttt{t} \equiv (\texttt{Node t1 t2})\,) & = & \mathit{leaves}(\texttt{Node t1 t2}) \\
 & = & \mathit{leaves}(\texttt{t1}) + \mathit{leaves}(\texttt{t2}) \\
(2\text{x IH}) & = & 2^{\mathit{depth}(\texttt{t1})} + 2^{\mathit{depth}(\texttt{t2})} \\
(\mathit{depth}(\texttt{t1}) = \mathit{depth}(\texttt{t2})) & = & 2^{\mathit{depth}(\texttt{t1})} + 2^{\mathit{depth}(\texttt{t1})} \\
 & = & 2 * 2^{\mathit{depth}(\texttt{t1})} \\
 & = & 2^{\mathit{depth}(\texttt{t1+1})} \\
 & = & 2^{\mathit{depth}(\texttt{t})}
\end{array}
$$

$\square$

Homework

Prove by means of structural induction:

Lemma 6.3.1.2

Let *depth*(`t`) and *leaves*(`t`) denote the depth and the number of leaves of any finite tree value `t :: Tree`, respectively. Then:

$$(\forall\, \texttt{t} :: \texttt{Tree}.\ \texttt{t}\ \mathit{finite}).\ \mathit{leaves}(\texttt{t}) \ \leq\ 2^{\mathit{depth}(\texttt{t})}$$

Note

...structural induction boils down to proof by cases if a data type is non-recursively defined.

```
Maybe a = Nothing | Just a

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just m) = f m
```

A value `x :: Maybe a` is called defined, if `x` equals `Nothing` or `x` equals `(Just m)` and `m :: a` is defined (cp. Chapter 6.3.2, why we are cautious on the value of `x`).

Lemma 6.3.1.3

$(\forall\, \mathtt{x :: Maybe\ Int}.\ \mathtt{x}\ \textit{defined})\,.\ \mathtt{maybe\ 2\ abs\ x} \geq 0$

Proof of Lemma 6.3.1.3

Case 1: Let `x` $\equiv$ `Nothing`. We obtain:

```
   maybe 2 abs x
=  maybe 2 abs Nothing
=  2
≥  0
```

Case 2: Let `x` $\equiv$ `Just m`, `m` defined. We obtain:

```
   maybe 2 abs x
=  maybe 2 abs (Just m)
=  abs m
≥  0
```

$\square$

Chapter 6.3.2

Inductive Proofs on Haskell Lists

Defined and Undefined Values

A computation which

- is faulty, i.e., produces an error or
- fails to (regularly) terminate

does not yield a proper value.

The value of such a computation is called

- undefined, or the undefined value

which is usually denoted by the symbol

- $\perp$ (read 'bottom').

Conversely, a properly terminating computation yields a value different from $\perp$, which is called

- defined or a defined value.

Example

The function

```
buggy_div :: Int -> Int
buggy_div n = div n 0
```

...produces an error for every argument called with.

The function

```
buggy_fac :: Int -> Int
buggy_fac n = (n-1) * buggy_fac n
buggy_fac 0 = 1
```

...fails to (regularly) terminate for every argument called with.

Very Simple Haskell Terms

...with a value equal to $\perp$:

- Error: The Prelude definition

  ```
  undefined :: a                    -- polymorphic
  undefined | False = undefined
  ```

  undefined ->> 'error' $\widehat{=}$ $\perp$

  is a very simple expression (of arbitrary type) whose evaluation always leads to an error due to case exhaustion.

- Non-termination: The co-recursive definition

  ```
  loop :: a                         -- polymorphic
  loop = loop
  ```

  loop ->> loop ->> loop ->> ... $\widehat{=}$ $\perp$

  is a very simple expression (of arbitrary type) whose evaluation leads to a non-terminating computation.

The Undefined Value $\perp$

...is a value of every Haskell data type representing the value of

- faulty or non-terminating computations.

Intuitively, $\perp$ can be considered the 'least accurate' approximation of (ordinary) values of the corresponding data type.

Informally

Lists are

- possibly empty finite sequences of values of the same type.

Haskell lists are

- built from the empty list.
  Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...
- composed of defined and undefined values.
  Examples: `[]`, `(1:2:[])`, `(1:`$\perp$`:3:[])`, `(`$\perp$`:`$\perp$`:3:[])`,...

Haskell lists are called

- defined, if all values are defined, i.e., different from $\perp$.
  Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...
  Counter-examples: `(`$\perp$`:[])`,`(1:`$\perp$`:[])`, `(`$\perp$`:2:`$\perp$`:[])`,...
- lists with undefined values, if at least one value is equal to the undefined value.
  Examples: `(`$\perp$`:[])`,`(1:`$\perp$`:[])`, `(`$\perp$`:2:`$\perp$`:[])`,...

Defined Lists, Lists with Undefined Values

Definition 6.3.2.1 (Defined, Undefined Values)

A value of a data type is called defined, if it is not equal to $\perp$; it is called undefined otherwise.

Definition 6.3.2.2 (List)

A list is a possibly empty finite sequence of (defined or undefined) values of the same type built from the empty list `[]`.

Definition 6.3.2.3 (Def. List, List w/ Undef. Values)

A list is called

- defined, if all its values are defined.
- a list with possibly undefined values, if some of its values can be equal to $\perp$.

Structural Induction for Defined Lists

Let P be a property on defined lists.

Proof pattern of structural induction for defined lists

1. Base case: Prove that $P(\texttt{[]})$ is true.
2. Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\texttt{x:xs})$ is true (induction step).

Note: The above pattern is an instance of the more general pattern of structural induction, specialized for defined lists.

Example 1: Induction over Defined Lists

Let

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Lemma 6.3.2.4

We have:

$$(\forall\, \mathtt{xs}, \mathtt{ys} :: \mathtt{[a]}.\ \mathtt{xs}, \mathtt{ys}\ \textit{defined}).$$
$$\mathtt{length\ (xs +\!+\ ys)} = \mathtt{length\ xs\ +\ length\ ys}$$

Proof by induction on the structure of xs.

Proof of Lemma 6.3.2.4 (1)

Let `ys :: [a]` be a defined list.

Base case: Let $\texttt{xs} \equiv \texttt{[]}$. As desired, we obtain by means of equational reasoning:

$$
\begin{aligned}
& \texttt{length (xs ++ ys)} \\
= \; & \texttt{length ([] ++ ys)} \\
= \; & \texttt{length ys} \\
= \; & \texttt{0 + length ys} \\
= \; & \texttt{length [] + length ys} \\
= \; & \texttt{length xs + length ys}
\end{aligned}
$$

Proof of Lemma 6.3.2.4 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{xs}$ defined. This implies $\texttt{xs}'$ (and $\texttt{x}$) is defined, too. By means of the induction hypothesis (IH), we can thus assume $\texttt{length (xs}' \texttt{++ ys)} = (\texttt{length xs}' \texttt{ + length ys)}$. This allows to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \texttt{length (xs ++ ys)} \\
 & = & \texttt{length ((x:xs}') \texttt{ ++ ys)} \\
 & = & \texttt{length (x:(xs}' \texttt{++ ys))} \\
 & = & \texttt{1 + length (xs}' \texttt{++ ys)} \\
\text{(IH)} & = & \texttt{1 + (length xs}' \texttt{ + length ys)} \\
 & = & \texttt{(1 + length xs}'\texttt{) + length ys} \\
 & = & \texttt{length (x:xs}'\texttt{) + length ys} \\
 & = & \texttt{length xs + length ys}
\end{array}
$$

□

Example 2: Induction over Defined Lists

Let

```
listSum :: Num a => [a] -> a
listSum []     = 0
listSum (x:xs) = x + listSum xs
```

Lemma 6.3.2.5

We have:

$$(\forall\, \texttt{xs} :: \texttt{[a]}.\ \texttt{xs}\ \textit{defined}).\ \texttt{listSum xs} = \texttt{foldr (+) 0 xs}$$

Proof by induction on the structure of xs.

Proof of Lemma 6.3.2.5 (1)

Base case: Let `xs` $\equiv$ `[]`. Equational reasoning yields the desired equality:

```
    listSum xs
=   listSum []
=   0
=   foldr (+) 0 []
=   foldr (+) 0 xs
```

Proof of Lemma 6.3.2.5 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{xs}$ defined. This implies $\texttt{xs}'$ (and $\texttt{x}$) is defined, too. By means of the induction hypothesis (IH), we can thus assume $\texttt{listSum xs}' = \texttt{foldr (+) 0 xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \texttt{listSum xs} \\
 & = & \texttt{listSum (x:xs}'\texttt{)} \\
 & = & \texttt{x + listSum xs}' \\
\text{(IH)} & = & \texttt{x + foldr (+) 0 xs}' \\
 & = & \texttt{foldr (+) 0 (x:xs}'\texttt{)} \\
 & = & \texttt{foldr (+) 0 xs}
\end{array}
$$

□

Example 3

Lemma 6.3.2.6

For all defined lists `xs :: [a]`, we have:

```
reverse (reverse xs) = xs
```

Proof by induction on the structure of `xs`.

Proof of Lemma 6.3.2.6 (1)

Base case: Let $\texttt{xs} \equiv \texttt{[]}$. Equational reasoning yields the desired equality:

$$
\begin{array}{rcl}
 & & \texttt{reverse (reverse xs)} \\
 & = & \texttt{reverse (reverse [])} \\
(\text{Def. }\texttt{reverse}) & = & \texttt{reverse []} \\
(\text{Def. }\texttt{reverse}) & = & \texttt{[]} \\
 & = & \texttt{xs}
\end{array}
$$

Proof of Lemma 6.3.2.6 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{xs}$ defined. This implies $\texttt{xs}'$ and $\texttt{x}$ are defined, too. By means of the induction hypothesis (IH), we can thus assume $\texttt{reverse (reverse xs}') = \texttt{xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
\texttt{reverse (reverse xs)} & = & \texttt{reverse (reverse (x:xs}'\texttt{))} \\
\text{(Def. }\texttt{reverse}\text{)} & = & \texttt{reverse ((reverse xs}'\texttt{) ++ [x])} \\
\text{(L. 6.3.2.8(1))} & = & \texttt{reverse [x] ++ reverse (reverse xs}'\texttt{)} \\
(\texttt{[x]} = \texttt{x:[]}\text{, IH}) & = & \texttt{reverse (x : []) ++ xs}' \\
\text{(Def. }\texttt{reverse}\text{)} & = & \texttt{(reverse [] ++ [x]) ++ xs}' \\
\text{(Def. }\texttt{reverse}\text{)} & = & \texttt{([] ++ [x]) ++ xs}' \\
\text{(Def. }\texttt{(++)}\text{)} & = & \texttt{[x] ++ xs}' \\
(\texttt{[x]} = \texttt{x:[]}) & = & \texttt{(x : []) ++ xs}' \\
\text{(Def. }\texttt{(++)}\text{)} & = & \texttt{x : ([] ++ xs}'\texttt{)} \\
\text{(Def. }\texttt{(++)}\text{)} & = & \texttt{x:xs}' \\
 & = & \texttt{xs}
\end{array}
$$

$\square$

Example 4

...sometimes, a truly inductive argument is not required.

Lemma 6.3.2.7

Let `f` be a strict map. Then:

$(\forall$ `xs :: [a]` . `xs` *defined*$)$. `(f . head) xs = head . (map f xs)`

Proof by cases.

Proof of Lemma 6.3.2.7 (1)

Case 1: Let $\texttt{xs} \equiv \texttt{[]}$. We get:

$$
\begin{array}{rcl}
 & & \texttt{(f . head) xs} \\
 & = & \texttt{(f . head) []} \\
\text{(Def. of } \texttt{(.)}\text{)} & = & \texttt{f (head [])} \\
\text{(Def. of } \texttt{head}\text{)} & = & \texttt{f } \bot \\
\text{(} \texttt{f} \text{ strict)} & = & \bot \\
\text{(Def. of } \texttt{head}\text{)} & = & \texttt{head []} \\
\text{(Def. of } \texttt{map}\text{)} & = & \texttt{head (map f [])} \\
\text{(Def. of } \texttt{(.)}\text{)} & = & \texttt{(head . map f) []} \\
 & = & \texttt{(head . map f) xs}
\end{array}
$$

Proof of Lemma 6.3.2.7 (2)

Case 2: Let $\texttt{xs} \equiv \texttt{(x:xs}'\texttt{)}$, $\texttt{xs}$ defined. This implies $\texttt{xs}'$ and $\texttt{x}$ are defined, too. We get:

$$
\begin{array}{rcl}
 & & \texttt{(f . head) xs} \\
 & = & \texttt{(f . head) (x:xs}'\texttt{)} \\
\text{(Def. of } \texttt{(.)}\text{)} & = & \texttt{f (head (x:xs}'\texttt{))} \\
\text{(Def. of } \texttt{head}\text{)} & = & \texttt{f x} \\
\text{(Def. of } \texttt{head}\text{, lazy eval.)} & = & \texttt{head (f x : map f xs}'\texttt{)} \\
\text{(Def. of } \texttt{map}\text{)} & = & \texttt{head (map f (x:xs}'\texttt{))} \\
\text{(Def. of } \texttt{(.)}\text{)} & = & \texttt{(head . map f) (x:xs}'\texttt{)} \\
 & = & \texttt{(head . map f) xs}
\end{array}
$$

$\square$

Note: The induction hypothesis $\texttt{(f . head) xs}' = \texttt{(head . map f) xs}'$ is not required to complete the proof of case 2; the inductive proof boils down to a proof by cases.

Homework (1)

...examples involving list reversions and concatenations.

Prove by means of structural induction on defined lists:

Lemma 6.3.2.8

For all defined lists `xs, ys, zs :: [a]`, we have:

1. `reverse (xs ++ ys)` $=$ `reverse ys ++ reverse xs`
2. `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
3. `xs ++ [] = xs`
4. `head (reverse xs)` $=$ `last xs`
5. `last (reverse xs)` $=$ `head xs`

Corollary 6.3.2.9

For all defined lists `xs :: [a]`, we have:

`xs ++ [] = xs = [] ++ xs`

Homework (2)

...examples involving list take and drop operations.

Prove by means of structural induction on defined lists:

Lemma 6.3.2.10

For all defined lists `xs :: [a]`, for all `m,n` $\in \mathbb{N}$, `m,n` ≥ 0, we have:

1.
```
take n xs ++ drop n xs = xs
take m . take n        = take (min m n)
drop m . drop n        = drop (m+n)
take m . drop n        = drop n . take (m+n)
```
2. If (additionally) `n` $\geq$ `m`, we have:
```
drop m . take n = take (n-m) . drop m
```

Homework (3)

...examples involving list foldings.

Prove by means of structural induction over defined lists:

Lemma 6.3.2.11

Let `op :: (a -> a -> a)` be associative with unit `e :: a`, i.e., $\forall$ `x :: a. e `op` x = x` $\wedge$ `` x `op` e = x ``. Then:

(∀ `xs :: [a]`. `xs` *defined*) . `foldr op e xs`
`= foldl op e xs`

Lemma 6.3.2.12

Let `op :: (a -> b -> b)` be an operator, `e :: b` a value. Then:

(∀ `xs :: [a]`. `xs` *defined*) . `foldr op e xs`
`= foldl (flip op) e (reverse xs)`

Homework (4)

...examples involving list foldings.

Prove by means of structural induction on defined lists:

Lemma 6.3.2.13

Let `op1, op2 :: (a -> a -> a)` be two operators, `e :: b` a value such that

```
∀ x,y,z :: a. x `op1` (y `op2` z) = (x `op1` y) `op2` z ∧
              x `op1` e = e `op2` x
```

Then:

(∀ `xs :: [a]. xs` *defined*) `. foldr op1 e xs`

`= foldl op2 e xs`

Homework (5)

...examples involving sequential composition and mappings.

Prove by means of structural induction on defined lists:

Lemma 6.3.2.14

1. `map (f . g)         = map f . map g`
2. `(map f) . tail      = tail . map f`
3. `(map f) . reverse   = reverse . map f`
4. `(map f) . concat    = concat . map (map f)`
5. `map f (xs ++ ys)    = map f xs ++ map f ys`
6. `map (\x -> x)       = \y -> y`

What are the types of the two anonymous λ-abstractions in Lemma 6.3.2.14(6)? Do they have the same type or different ones?

Structural Induction for Lists w/ Undef. Values

Let P be a property on lists with possibly undefined values.

Proof pattern of structural induction for lists with possibly undefined values:

1. Base case: Prove that $P(\texttt{[]})$ is true.
2. Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\perp\texttt{:xs})$ and $P(\texttt{x:xs})$, $\texttt{x}$ a defined value, are true (induction step).

Note: The above pattern is an instance of the more general pattern of structural induction, specialized for lists with possibly undefined values.

Example: Induct. over Lists w/ Undef. Values

Let

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Lemma 6.3.2.15

We have:

$(\forall\, \mathtt{xs}, \mathtt{ys} :: \mathtt{[a]}.\ \mathtt{xs}, \mathtt{ys}$ *lists w/ possibly undefined values*$)$.

$$\mathtt{length}\ (\mathtt{xs} +\!\!+ \mathtt{ys}) = \mathtt{length}\ \mathtt{xs}\ \mathtt{+}\ \mathtt{length}\ \mathtt{ys}$$

Proof by induction on the structure of xs.

Proof of Lemma 6.3.2.15 (1)

Let `ys :: [a]` be a list with possibly undefined values.

Base case: Let $\texttt{xs} \equiv \texttt{[]}$. As desired, we obtain by means of equational reasoning:

```
     length (xs ++ ys)
  =  length ([] ++ ys)
  =  length ys
  =  0 + length ys
  =  length [] + length ys
  =  length xs + length ys
```

Proof of Lemma 6.3.2.15 (2)

Inductive case 1: Let $\texttt{xs} \equiv (\perp : \texttt{xs}')$. By means of the induction hypothesis (IH), we can assume $\texttt{length (xs}' \mathbin{++} \texttt{ys)} = \texttt{(length xs}' \texttt{ + length ys)}$. This allows to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \texttt{length (xs ++ ys)} \\
 & = & \texttt{length (}(\perp : \texttt{xs}')\texttt{ ++ ys)} \\
 & = & \texttt{length (}\perp : (\texttt{xs}' \mathbin{++} \texttt{ys}))\\
 & = & \texttt{1 + length (xs}' \mathbin{++} \texttt{ys)} \\
\text{(IH)} & = & \texttt{1 + (length xs}' \texttt{ + length ys)} \\
 & = & \texttt{(1 + length xs}'\texttt{) + length ys} \\
 & = & \texttt{length (}\perp : \texttt{xs}'\texttt{) + length ys} \\
 & = & \texttt{length xs + length ys}
\end{array}
$$

Proof of Lemma 6.3.2.15 (3)

Inductive case 2: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{x}$ defined. By means of the induction hypothesis (IH), we can assume $\texttt{length (xs}' \texttt{++ ys)} = \texttt{(length xs}' \texttt{ + length ys)}$. This allows to complete the proof as follows:

$$
\begin{array}{lcl}
 & & \texttt{length (xs ++ ys)} \\
 & = & \texttt{length ((x:xs}'\texttt{) ++ ys)} \\
 & = & \texttt{length (x:(xs}'\texttt{ ++ ys))} \\
 & = & \texttt{1 + length (xs}'\texttt{ ++ ys)} \\
\text{(IH)} & = & \texttt{1 + (length xs}'\texttt{ + length ys)} \\
 & = & \texttt{(1 + length xs}'\texttt{) + length ys} \\
 & = & \texttt{length (x:xs}'\texttt{) + length ys} \\
 & = & \texttt{length xs + length ys}
\end{array}
$$

$\square$

Homework

Which of the statements of the lemmas of Chapter 6.3.2 hold for lists with possibly undefined values, too?

Prove your claims or provide counter-examples.

Chapter 6.3.3

Inductive Proofs on Partial Haskell Lists

Informally

Haskell lists are called

- partial, if they are built from the undefined list.
  Examples: $\bot$, `(1:`$\bot$`)`, `(1:2:3:`$\bot$`)`, `(1:2:3:`$\bot$`)`,...
- partial with possibly undefined values, if they are partial and at least one of their values is equal to the undefined value.
  Examples: `(1:`$\bot$`:3:`$\bot$`)`, `(1:`$\bot$`:3:`$\bot$`)`. `(`$\bot$`:`$\bot$`:3:`$\bot$`)`,...

Note the different types of $\bot$ and $\bot$ in the above examples:

$$\bot :: \texttt{Int}$$

$$\bot :: \texttt{[Int]}$$

Partial Lists

Definition 6.3.3.1 (Partial List)

A partial list is a possibly empty finite sequence of (defined or undefined) values of the same type built from the undefined list $\perp$.

Definition 6.3.3.2 (Defined Part. List, Part. List w/ Undef. Values)

A partial list is called

- defined, if all its values are defined.
- a partial list with possibly undefined values, if some of its values can be equal to $\perp$.

Examples

...of lists and partial lists w/ and w/out undefined values:

```
empty = []                      -- Empty list
ns = 2 : 3 : 5 : 7 : []         -- Defined list
ms = 2 : loop : 5 : 7 : []      -- List w/ undefined
                                -- values

pempty = loop :: [Int]          -- Empty partial list
xs = 2 : 3 : 5 : 7 : loop       -- Def. partial list
ys = 2 : loop : 5 : 7 : loop    -- Partial list w/
                                -- undefined values
```

Note: The value of all occurrences of `loop` in `ns`, `ms`, `xs`, `ys`, and `pempty` is equal to $\perp$ but of different type:

- `loop :: Int` in `ms` and `ys`.
- `loop :: [Int]` in `pempty`, `xs`, and `ys` .

Evaluating Terms w/, w/out Undef. Values (1)

...using `reverse`, `head`, `tail`, and `last` as examples:

```
reverse ns ->> [7,5,3,2]
reverse ms ->> [7,5  ...followed by an infinite wait
reverse xs ->> ...infinite wait
reverse ys ->> ...infinite wait

head (reverse ms) ->> 7          -- thanks to lazy eval.
head (tail (reverse ms)) ->> 5  -- thanks to lazy eval.
head (tail (tail (reverse ms))) ->> ...infinite wait

head (tail (reverse xs)) ->> ...infinite wait

last ms ->> 7
last xs ->> ...infinite wait

reverse (reverse ms) ->> [2 ...followed by an
                                    infinite wait
head (reverse (reverse ms)) ->> 2
```

Evaluating Terms w/, w/out Undef. Values (2)

...using `length` and `take` as examples:

```
length ns ->> 4
length ms ->> 4
length xs ->> ...infinite wait
length ys ->> ...infinite wait

length (take 4 ns) ->> 4
length (take 3 ms) ->> 3
length (take 2 xs) ->> 2
length (take 3 ys) ->> 3

length (take 5 ns) ->> 4
length (take 4 xs) ->> 4
length (take 5 xs) ->> ...infinite wait
```

Note (1)

...the different behaviour is due to making or not-making a pattern match on the values of the argument list by length, reverse, take, drop, head, tail, and last, respectively:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs          -- No pattern match
                                       -- on the head of the
                                       -- argument list!

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x] -- Pattern match on
                                   -- the head of the
                                   -- argument list!

reverse :: [a] -> [a]
reverse = foldl (flip (:)) []      -- Same here, even if
                                   -- pointfree defined!
```

Note (2)

...the definitions of take and drop recalled:

```
take :: Int -> [a] -> [a]
take n _    |  n <= 0  = []
take _ []              = []
take n (x:xs)          = x : take (n-1) xs -- Pattern
                                  -- match on the head of
                                  -- of the argument list!

drop :: Int -> [a] -> [a]
drop n xs   |  n <= 0  = xs
drop _ []              = []                -- No pattern
drop n (_:xs)          = drop (n-1) xs     -- match on the
                                           -- head of the
                                           -- argument list!
```

Note (3)

...the definitions of `head` and `last` recalled:

```
head :: [a] -> a
head (x:_) = x                   -- Pattern match on the
                                 -- head of the argu-
                                 -- ment list!

tail :: [a] -> [a]
tail (_:xs) = xs                 -- No pattern match
                                 -- on the head of the
                                 -- argument list!

last :: [a] -> a
last [x]    = x                  -- Pattern match on the
last (_:xs) = last xs            -- argument list
```

Note (4)

...data and newtype declarations behave differently regarding pattern matching. Consider:

```
data Bool′ = B′ Bool

hello′ :: Bool′ -> String
hello′ (B′ _) = "Hello!"

hello′ loop ->> ...infinite wait  -- Pattern match
             -- required since data declarations might
             -- have more than one data constructor.

newtype Bool″ = B″ Bool

hello″ :: Bool″ -> String
hello″ (B″ _) = "Hello!"

hello″ loop ->> "Hello!" -- No Pattern match required
                         -- since newtype declarations have
                         -- exactly one data constructor.
```

Note (5)

...the following variant of `hello'` behaves differently, since pattern matching is no longer required:

```
data Bool' = B' Bool

hello''' :: Bool' -> String
hello''' _ = "Hello!"

hello''' loop ->> "Hello!" -- No Pattern match required
                           -- since any of possibly
                           -- many data constructors
                           -- matches.
```

In summary: Undefined values cause program failure, whenever they need to be (partially) evaluated for pattern matching or to be displayed as (part of) the result of evaluating a term; the details are subtle as demonstrated by the examples.

The Inductive Proof Patterns

...introduced in Chapter 6.3.2 apply to

- defined lists
- lists with possibly undefined values

which are built by definition from the empty list `[]`.

By contrast, partial lists are built from the undefined list $\perp$ (such as `xs`) and may contain values equal to the undefined value (such as `ys`).

We thus need a new inductive proof principle tailored for partial lists (with possibly undefined values).

Inductive Proofs on Partial Lists

Let P be a property defined on partial lists.

Proof pattern for defined partial lists:

1. Base case: Prove that $P(\bot)$ is true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\mathtt{x}:\mathtt{xs})$ is true (induction step).

Proof pattern for partial lists with possibly undefined values:

1. Base case: Prove that $P(\bot)$ is true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\bot:\mathtt{xs})$ and $P(\mathtt{x}:\mathtt{xs})$, $\mathtt{x}$ a defined value, are true (induction step).

Homework (1)

Does Lemma 6.3.2.7 recalled below hold for defined partial lists, too? Does it make a difference if partial lists may have values equal to the undefined value or not?

Lemma 6.3.2.7

Let `f` be a strict map. Then:

$(\forall\, \texttt{xs :: [a]}.\ \texttt{xs}\ \textit{defined}).\ \texttt{(f . head) xs = head . (map f xs)}$

Provide a proof or a counter-example to support your claims.

Homework (2)

Which of the statements of the lemmas in Chapter 6.3.2 hold for

- defined partial lists?
- partial lists with possibly undefined values?

Prove your claims or provide counter-examples.

Inductive Proofs on Lists and Partial Lists

Let P be a property defined on lists and partial lists.

Proof pattern for lists and partial lists with possibly undefined values:

1. Base case: Prove that $P(\perp)$ and $P([\,])$ are true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\perp\mathtt{:xs})$ and $P(\mathtt{x:xs})$, $\mathtt{x}$ a defined value, are true (induction step).

Homework

Which of the statements of the lemmas in Chapter 6.3.2 and 6.3.3 hold for

- defined lists and defined partial lists?
- lists and partial lists with possibly undefined values?

Prove your claims or provide counter-examples.

Chapter 6.3.4

Inductive Proofs on Haskell Stream Approximants

Streams

...are infinite sequences of values of the same type.

Definition 6.3.4.1 (Stream)

A stream is an infinite sequence of (defined or undefined) values of the same type.

Definition 6.3.4.2 (Def. Stream, S. w/ Undef. Values)

A stream is called

- defined, if all its values are defined.
- a stream with possibly undefined values, if some of its values can be equal to $\perp$.

Homework: Is it meaningful to say, a stream were built from the empty or the undefined stream?

Comparing Partial Lists: Approximation Order

...intuitively, a partial list `xs` approximates a partial list `ys`, if `xs` is 'equal to but less defined' than `ys`, $\texttt{xs} \sqsubseteq \texttt{ys}$:

$$
\begin{array}{rcl}
\bot & \sqsubseteq & 0 : \bot \\
0 : \bot & \sqsubseteq & 0 : 1 : \bot \\
0 : 1 : \bot & \sqsubseteq & 0 : 1 : 1 : \bot \\
0 : 1 : 1 : 2 : \bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : \bot \\
 & \ldots & \\
\bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot \\
0 : \bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot \\
0 : 1 : 2 : \bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot \\
 & \ldots &
\end{array}
$$

Streams can be approximated by infinite sequences of

- increasingly more accurate partial lists, called PL-approximants.

Illustrating Stream Approximation

...the stream of natural numbers

`[1..] = 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 :...`

is approximated by the infinite sequence of more and more accurate PL-approximants, whose limit is the stream itself:

$$
\begin{array}{l}
\quad\;\bot \\
\sqsubseteq\ 1 : \bot \\
\sqsubseteq\ 1 : 2 : \bot \\
\sqsubseteq\ 1 : 2 : 3 : \bot \\
\sqsubseteq\ 1 : 2 : 3 : 4 : \bot \\
\sqsubseteq\ 1 : 2 : 3 : 4 : 5 : \bot \\
\sqsubseteq\ 1 : 2 : 3 : 4 : 5 : 6 : \bot \\
\sqsubseteq\ 1 : 2 : 3 : 4 : 5 : 6 : 7 : \bot \\
\sqsubseteq\ 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \bot \\
\ldots \\
\sqsubseteq\ 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots = [1..]
\end{array}
$$

Intuitively

...the undefined list $\perp$ is the 'least defined,' hence the 'least accurate' partial list (or approximant). Sequences of more and more 'defined' approximants are getting more and more accurate.

...considering (finite) partial lists

- approximations, called approximants, of streams equals in spirit the approach of outputting/printing a stream prefix by interrupting the printing of the stream after some period of time by hitting `Ctrl-C`.

Extending this period of time further and further yields

- successively more accurate approximants of the stream.

Approximation Order on Partial Lists, Streams

...formalizing the idea of approximation:

Definition 6.3.4.3 (Partially Ordered Set)

A binary relation R on M is called a partially ordered set (or partial order) iff R is reflexive, transitive, and anti-symmetric; the pair (M, R) is called a partial order.

Let $S_{(PL,St)} =_{df} \{s \mid s \text{ partial list or stream}\}$ be the set of partial lists and streams.

Lemma 6.3.4.4 (Approximation Order)

The relation $\sqsubseteq$ on $S_{(PL,St)}$ defined by:

$$\begin{array}{lcll} \bot & \sqsubseteq & \mathtt{xs} & \\ \mathtt{x}\ :\ \mathtt{xs} & \sqsubseteq & \mathtt{y}\ :\ \mathtt{ys} & \Longleftrightarrow_{df}\ \mathtt{x} \equiv \mathtt{y}\ \wedge\ \mathtt{xs} \sqsubseteq \mathtt{ys} \end{array}$$

is a partial order, called approximation order, where $\equiv$ denotes equality on partial list/stream elements.

Partial Lists as Stream Approximants

Definition 6.3.4.5 (PL-Approximants)

The set of PL-approximants of a defined stream `xs` is defined by $\mathit{PL\text{-}Approx}(\mathtt{xs}) =_{df} \{\, \mathtt{take'}\ \mathtt{n}\ \mathtt{xs} \mid \mathtt{n} \in \mathbb{N}_0 \,\}$, where

```
take' :: Integer -> [a] -> [a]
take' n _        | n <= 0  = undefined
take' n (x:xs)             = x : take' (n-1) xs
```

Note: PL-approximants are built from the undefined list, not the empty list; they all have finite length.

Examples:

- $\mathit{PL\text{-}Approx}(\mathtt{[1\,..]}) = \{\bot, \mathtt{1}:\bot, \mathtt{1}:\mathtt{2}:\bot, \mathtt{1}:\mathtt{2}:\mathtt{3}:\bot, \ldots\}$
- $\mathit{PL\text{-}Approx}(\mathtt{[1,1\,..]}) = \{\bot, \mathtt{1}:\bot, \mathtt{1}:\mathtt{1}:\bot, \mathtt{1}:\mathtt{1}:\mathtt{1}:\bot, \ldots\}$

Chains, Domains

Definition 6.3.4.6 (Chain)

A subset $C \subseteq P$ of a partial order $(P, \sqsubseteq)$ is called a chain, if the elements of C are totally ordered.

Definition 6.3.4.7 (Domain)

A partial order $(D, \sqsubseteq)$ is called a domain (or complete partial order (CPO)), if

1. D has a least element $\perp$.
2. $\bigsqcup C$ exists for every chain C in D.

$\sqsubseteq$ is then called approximation order of $(D, \sqsubseteq)$.

Example: Let $\mathcal{P}(\mathbb{N})$ be the power set of $\mathbb{N}$. Then: $(\mathcal{P}(\mathbb{N}), \sqsubseteq)$, $\sqsubseteq =_{df} \subseteq$, is a domain with least element $\emptyset$ and $\bigsqcup C = \bigcup C$ for every chain $C \subseteq \mathcal{P}(\mathbb{N})$.

Main Results

Lemma 6.3.4.8 (Partial Lists and Streams Domain)

$(S_{(PL,St)}, \sqsubseteq)$ is a domain with the undefined list $\perp$ as least element, and the order $\sqsubseteq$ defined in Lemma 6.3.4.4 as approximation order.

Lemma 6.3.4.9 (PL-Approximants Chain)

The set $\mathit{PL\text{-}Approx}(\mathtt{xs})$ of a defined stream $\mathtt{xs}$ is a chain.

Theorem 6.3.4.10 (Approximation)

A defined stream $\mathtt{xs}$ is equal to the least upper bound of its PL-approximants set, also called its limit:

$$\bigsqcup \mathit{PL\text{-}Approx}(\mathtt{xs}) \;=\; \bigsqcup_{n=0}^{\infty} \mathtt{take'}\ \mathtt{n}\ \mathtt{xs} \;=\; \mathtt{xs}$$

Note: Refer to Appendix A for the definition of technical terms and illustrating examples, if required.

Streams as Limit of their PL-Approximants Sets

...the set of PL-approximants of a defined stream is a chain with the stream itself as its least upper bound (cf. Approximation Theorem 6.3.4.10) as illustrated below:

$$
\begin{array}{l}
\quad \bot \\
\sqsubseteq 1 : \bot \\
\sqsubseteq 1 : 2 : \bot \\
\sqsubseteq 1 : 2 : 3 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \bot \\
\ldots \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots = [1..]
\end{array}
$$

Finite and Infinite Sequences of Values

...are quite diverse objects enjoying different properties.

Properties valid for lists (i.e., finite sequences) might hold or might not hold for streams (i.e., infinite sequences) and vice versa, e.g.:

- $\forall z \in \mathbb{Z}.$ `take n xs ++ drop n xs = xs`
  ...does hold for defined lists and streams.
- `reverse (reverse xs)) = xs`
  ...does hold for defined lists but not for streams.
- $\forall n \in \mathbb{N}.$ `drop n xs` $\neq$ `[]`
  ...does hold for streams but not for lists.

Finite PL-Approximants and Streams

...are quite diverse objects, too.

Properties which are valid for every partial list of the infinite set of finite PL-approximants of a stream might hold or might not hold for its limit, the stream itself, and vice versa, e.g.:

- `map (f . g) xs = (map f . map g) xs`
  does hold for all PL-approximants of a defined stream and the stream itself.
- *'This sequence is partial'*
  ...does hold for all PL-approximants of a stream but not for the stream itself.
- `tail xs` *'is a stream'*
  ...does hold for a stream but not for any of its PL-approximants.

Reconsidering the Induction Principles

...considered so far.

The induction principles of Chapter 6.3.2 and 6.3.3 apply to

- finite sequences of (possibly undefined) values

and thus allow to prove properties for all finite lists and/or all finite partial lists (with possibly undefined values).

Streams, however, are by definition

- infinite sequences of values.

Thus, the induction principles of Chapter 6.3.2 and 6.3.3 are not applicable for free for proving properties on streams, especially in the light of the fact that properties being valid for all PL-approximants of a stream need not hold for the stream itself.

Fortunately

...the induction principle for partial lists (with and without possibly undefined values) of Chapter 6.3.3 can be used to prove so-called (in analogy to Definition 6.6.1) admissible properties for streams.

Intuitively, a property is admissible, if it holds for the limit of a PL-approximants set, if it holds for each of its elements.

Equational properties are admissible.

Together with Approximation Theorem 6.3.4.10, this justifies the proceeding considered next.

Inductive Proofs on PL-Approximants Sets

...for proving 'admissible' properties of streams.

Let P be an equational property defined on PL-approximants and streams.

Proof pattern for defined PL-approximants:

- Base case: Prove that $P(\perp)$ is true.
- Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\mathtt{x}:\mathtt{xs})$ is true (induction step).

Proof pattern for PL-approximants w/ possibly undef. values:

- Base case: Prove that $P(\perp)$ is true.
- Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\perp:\mathtt{xs})$ and $P(\mathtt{x}:\mathtt{xs})$, $\mathtt{x}$ a defined value, are true (induction step).

Example 1: Induction on PL-Approximants

Lemma 6.3.4.11

We have:

$$(\forall\, \mathtt{xs} \in \mathtt{[a]}\,.\ \mathtt{xs}\ \ \textit{defined}\ \ \textit{stream})\ \forall\, \mathtt{n} \in \mathbb{N}.$$
$$\mathtt{take\ n\ xs\ ++\ drop\ n\ xs\ =\ xs}$$

Proof by cases and induction on the structure of `xs`.

Proof of Lemma 6.3.4.11 (1)

Case 1: Let $n \in \mathbb{N}$, $n = 0$, and xs be some defined stream. Equational reasoning yields the desired equality:

$$
\begin{array}{rcl}
 & & \texttt{take n xs ++ drop n xs} \\
 & = & \texttt{take 0 xs ++ drop 0 xs} \\
\text{(Def. take)} & = & \texttt{[] ++ xs} \\
 & = & \texttt{xs}
\end{array}
$$

Case 2: Let $n \in \mathbb{N}$, $n \geq 1$ be some natural number. We now proceed by induction on the structure of xs.

Base case: Let $xs \equiv \perp$. Equational reasoning yields as desired:

$$
\begin{array}{rcl}
 & & \texttt{take n xs ++ drop n xs} \\
 & = & \texttt{take n } \perp \texttt{ ++ drop n } \perp \\
\text{(Def. take, case exh.)} & = & \perp \texttt{ ++ } \perp \\
 & = & \perp \\
 & = & \texttt{xs}
\end{array}
$$

Proof of Lemma 6.3.4.11 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$ be a defined PL-approximant. Then $\texttt{x}$ is defined and $\texttt{xs}'$ is a defined PL-approximant, too. By means of Case 1 (if $\texttt{n=1}$) and the induction hypothesis (IH) (if $\texttt{n}>\texttt{1}$), we can assume for all $n \in \mathbb{N}$ the equality $(\texttt{take (n-1) xs}' \texttt{ ++ drop (n-1) xs}') = \texttt{xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{lcl}
 & & \texttt{take n xs ++ drop n xs} \\
 & = & \texttt{take n } (\texttt{x:xs}') \texttt{ ++ drop n } (\texttt{x:xs}') \\
 & = & \texttt{x : } (\texttt{take (n-1) xs}' \texttt{ ++ drop (n-1) xs}') \\
\text{(Case 1, IH)} & = & \texttt{x : xs}' \\
 & = & (\texttt{x:xs}') \\
 & = & \texttt{xs}
\end{array}
$$

□

Example 2: Induction on PL-Approximants

Consider the following variant of Lemma 6.3.4.11:

Lemma 6.3.4.12

We have:

$$(\forall\, \texttt{xs} \in \texttt{[a]}\,.\ \texttt{xs}\ \ \textit{defined}\ \ \textit{stream})\ \forall\, \texttt{z} \in \mathbb{Z}\,.$$
$$\texttt{take z xs ++ drop z xs = xs}$$

Proof by induction on the structure of xs and cases.

Proof of Lemma 6.3.4.12 (1)

Base case: Let $\mathtt{xs} \equiv \bot$.

Case 1: Let $\mathtt{z} \in \mathbb{Z}$, $\mathtt{z} \le 0$. Equational reasoning yields the desired equality:

$$
\begin{aligned}
& \mathtt{take\ z\ xs\ ++\ drop\ z\ xs} \\
= \;& \mathtt{take\ z}\ \bot\ \mathtt{++\ drop\ z}\ \bot \\
= \;& \mathtt{[]\ ++}\ \bot \\
= \;& \bot \\
= \;& \mathtt{xs}
\end{aligned}
$$

Case 2: Let $\mathtt{z} \in \mathbb{Z}$, $\mathtt{z} > 0$. Again, equational reasoning yields as desired:

$$
\begin{aligned}
& \mathtt{take\ z\ xs\ ++\ drop\ z\ xs} \\
= \;& \mathtt{take\ z}\ \bot\ \mathtt{++\ drop\ z}\ \bot \\
= \;& \bot\ \mathtt{++}\ \bot \\
= \;& \bot \\
= \;& \mathtt{xs}
\end{aligned}
$$

Proof of Lemma 6.3.4.12 (2)

Inductive case: Let $\mathtt{xs} \equiv (\mathtt{x}:\mathtt{xs}')$ be a defined PL-approximant, and $\mathtt{z} \in \mathbb{Z}$. $\mathtt{xs}$ defined implies that $\mathtt{x}$ is defined and that $\mathtt{xs}'$ is a defined PL-approximant, too. By means of the induction hypothesis (IH), we can assume for all $z \in \mathbb{Z}$ the equality $(\mathtt{take}\ (\mathtt{z-1})\ \mathtt{xs}'\ \mathtt{++}\ \mathtt{drop}\ (\mathtt{z-1})\ \mathtt{xs}') = \mathtt{xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{lcl}
 & & \mathtt{take\ z\ xs\ ++\ drop\ z\ xs} \\
 & = & \mathtt{take\ z\ (x:xs')\ ++\ drop\ z\ (x:xs')} \\
 & = & \mathtt{x : (take\ (z-1)\ xs'\ ++\ drop\ (z-1)\ xs')} \\
\text{(IH)} & = & \mathtt{x : xs'} \\
 & = & \mathtt{(x:xs')} \\
 & = & \mathtt{xs}
\end{array}
$$

□

Example 3: Induction on PL-Approximants

Lemma 6.3.4.13

We have:

$(\forall\, \texttt{xs} \in \texttt{[a]}\,.\ \texttt{xs}\ \textit{defined stream})\,.$
$\qquad\qquad \texttt{map (f . g) xs = (map f . map g) xs}$

Proof by induction on the structure of `xs`.

Proof of Lemma 6.3.4.13 (1)

Base case: Let $\texttt{xs} \equiv \bot$. Equational reasoning yields the desired equality:

$$
\begin{array}{rcl}
 & & \texttt{map (f . g) xs} \\
 & = & \texttt{map (f . g)}\ \bot \\
\text{(Def. }\texttt{map}\text{, case exh.)} & = & \bot \\
\text{(Def. }\texttt{map}\text{, case exh.)} & = & \texttt{map f}\ \bot \\
\text{(Def. }\texttt{map}\text{, case exh.)} & = & \texttt{map f (map g}\ \bot\texttt{)} \\
\text{(Def. }\texttt{(.)}\text{)} & = & \texttt{(map f . map g)}\ \bot \\
 & = & \texttt{(map f . map g) xs}
\end{array}
$$

Proof of Lemma 6.3.4.13 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$ be a defined PL-approximant. Then $\texttt{x}$ is defined and $\texttt{xs}'$ is a defined PL-approximant, too. By means of the induction hypothesis (IH), we can assume the equality $\texttt{map (f . g) xs}' = \texttt{(map f . map g) xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \texttt{map (f . g) xs} \\
 & = & \texttt{map (f . g) (x:xs}'\texttt{)} \\
\text{(Def. map)} & = & \texttt{((f . g) x) : map (f . g) xs}' \\
\text{(IH)} & = & \texttt{((f . g) x) : (map f . map g) xs}' \\
\text{(2x Def. (.))} & = & \texttt{f (g x) : (map f (map g xs}'\texttt{))} \\
\text{(Def. map)} & = & \texttt{map f (g x : map g xs}'\texttt{)} \\
\text{(Def. (.))} & = & \texttt{map f (map g (x:xs}'\texttt{))} \\
 & = & \texttt{(map f . map g) (x:xs}'\texttt{)} \\
 & = & \texttt{(map f . map g) xs}
\end{array}
$$

□

Homework (1)

In Definition 6.3.4.5, the set of PL-approximants is defined for defined streams.

1. Extend the notion of PL-approximant sets to streams with possibly undefined values.
2. Adapt the definition of the approximation order $\sqsubseteq$ (cf. Lemma 6.3.4.4), the Approximation Theorem 6.3.4.10, and the inductive principle for PL-approximants sets accordingly.
3. Do Lemma 6.3.4.11, 6.3.4.12, and 6.3.4.13 hold for streams with possibly undefined values, too? Prove your claims or provide counter-examples.

Homework (2)

Consider Claim 6.3.2.6′, which extends the statement of Lemma 6.3.2.6 to defined streams, and the subsequent attempt to prove it. At first sight, the 'proof' attempt looks quite reasonable. Nonetheless, there must be a flaw. Which one? Where and why?

Claim 6.3.2.6′

For all defined streams `xs :: [a]`, we have:

```
reverse (reverse xs) = xs
```

'Proof' attempt by induction on the structure of `xs`.

‘Proof’ Attempt of Claim 6.3.2.6′ (1)

Base case: Let $\texttt{xs} \equiv \perp$. Equational reasoning yields the desired equality:

$$
\begin{array}{rcl}
 & & \texttt{reverse (reverse xs)} \\
 & = & \texttt{reverse (reverse } \perp\texttt{)} \\
\text{(Def. } \texttt{reverse}\text{, case exh.)} & = & \texttt{reverse } \perp \\
\text{(Def. } \texttt{reverse}\text{, case exh.)} & = & \perp \\
 & = & \texttt{xs}
\end{array}
$$

'Proof' Attempt of Claim 6.3.2.6′ (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{xs}$ defined. This implies $\texttt{xs}'$ and $\texttt{x}$ are defined, too. By means of the induction hypothesis (IH), we can thus assume $\texttt{reverse (reverse xs}') = \texttt{xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
\texttt{reverse (reverse xs)} & = & \texttt{reverse (reverse (x:xs}'\texttt{))} \\
\text{(Def. } \texttt{reverse}\text{)} & = & \texttt{reverse ((reverse xs}'\texttt{) ++ [x])} \\
\text{(L. 6.3.2.8(1))} & = & \texttt{reverse [x] ++ reverse (reverse xs}'\texttt{)} \\
(\texttt{[x]=x:[]}, \text{IH}) & = & \texttt{reverse (x : []) ++ xs}' \\
\text{(Def. } \texttt{reverse}\text{)} & = & \texttt{(reverse [] ++ [x]) ++ xs}' \\
\text{(Def. } \texttt{reverse}\text{)} & = & \texttt{([] ++ [x]) ++ xs}' \\
\text{(Def. } \texttt{(++)}\text{)} & = & \texttt{[x] ++ xs}' \\
(\texttt{[x]=x:[]}) & = & \texttt{(x : []) ++ xs}' \\
\text{(Def. } \texttt{(++)}\text{)} & = & \texttt{x : ([] ++ xs}'\texttt{)} \\
\text{(Def. } \texttt{(++)}\text{)} & = & \texttt{x:xs}' \\
 & = & \texttt{xs} \qquad \square
\end{array}
$$

Homework (3)

Recall that properties, which hold for (defined) lists

- can hold, e.g.,
  $\forall z \in \mathbb{Z}.$ `take n xs ++ drop n xs = xs`
- but need not hold, e.g.,
  `reverse (reverse xs)) = xs`

for (defined) streams.

Which of the statements of the lemmas in Chapter 6.3.2, 6.3.3, and 6.3.4 hold for

- defined streams?
- streams with possibly undefined elements?

Prove your claims or provide counter-examples.

Approximation Order on Lists, Part. Lists, Streams

Let $S_{(L,PL,St)} =_{df} \{s \mid s \text{ list or partial list or stream}\}$ be the set of lists, partial lists and streams.

Lemma 6.3.4.14 (Approximation Order)

The relation $\sqsubseteq$ on $S_{(L,PL,St)}$ defined by:

$$
\begin{array}{lllll}
\bot & \sqsubseteq & \texttt{xs} & & \\
[\,] & \sqsubseteq & \texttt{xs} & \Longleftrightarrow_{df} & xs = [\,] \\
\texttt{x} : \texttt{xs} & \sqsubseteq & \texttt{y} : \texttt{ys} & \Longleftrightarrow_{df} & \texttt{x} \equiv \texttt{y} \wedge \texttt{xs} \sqsubseteq \texttt{ys}
\end{array}
$$

is a partial order, called approximation order.

Partial Lists as List and Stream Approximants

Definition 6.3.4.15 (LPL-Approximants)

The set of LPL-approximants of a defined stream xs is defined by $\mathit{LPL\text{-}Approx}(\texttt{xs}) =_{df} \{\, \texttt{approx n xs} \mid \texttt{n} \in \mathbb{N}_0 \,\}$, where

```
approx :: Integer -> [a] -> [a]
approx (n+1) []     = []
approx (n+1) (x:xs) = x : approx n xs
```

Note: There are LPL-approximants built from the undefined list and others built from empty list; they all have finite length.

Examples:

- $\mathit{LPL\text{-}Approx}(\perp) = \{\perp\}$
- $\mathit{LPL\text{-}Approx}(\texttt{[]}) = \{\perp,\ \texttt{[]}\}$
- $\mathit{LPL\text{-}Approx}(\texttt{[1,2,3]}) = \{\perp, \texttt{1:}\perp, \texttt{1:2:}\perp, \texttt{1:2:3:[]}\}$
- $\mathit{LPL\text{-}Approx}(\texttt{[1..]}) = \{\perp, \texttt{1:}\perp, \texttt{1:2:}\perp, \texttt{1:2:3:}\perp, \ldots\}$
- $\mathit{LPL\text{-}Approx}(\texttt{[1,1..]}) = \{\perp,\ \texttt{1:}\perp, \texttt{1:1:}\perp,\ \texttt{1:1:1:}\perp, \ldots\}$

Note

...approx is similar to take′ used in Definition 6.3.4.5, however, behaves differently when applied to lists (which, by definition, are built from the empty list, not the undefined list):

```
approx :: Integer -> [a] -> [a]
approx (n+1) []     = []
approx (n+1) (x:xs) = x : approx n xs
```

Note: Pattern n+1 matches only positive integers ≥ 1. Thus:

1. approx m ys ->> ys,
   if m $>$ len ys.
2. approx m ys ->> y_0 : y_1 : ... : y_{m-1} : $\perp$,
   if m $\leq$ len ys, i.e., approx will cause an error after generating the first m elements of ys.

Examples

```
approx 0 [1,2]    ->> ⊥
approx 1 [1,2]    ->> approx (0+1) [1,2]
                  ->> 1 : approx 0 [2]
                  ->> 1 : ⊥
approx 2 [1,2]    ->> approx (1+1) [1,2]
                  ->> 1 : approx 1 [2]
                  ->> 1 : approx (0+1) [2]
                  ->> 1 : 2 : approx 0 []
                  ->> 1 : 2 :  ⊥
approx 3 [1,2]    ->> approx (2+1) [1,2]
                  ->> 1 : approx 2 [2]
                  ->> 1 : approx (1+1) [2]
                  ->> 1 : 2 : approx 1 []
                  ->> 1 : 2 : approx (0+1) []
                  ->> 1 : 2 : []
approx 7 [1,2..]  ->> 1 : 2 : 3 : 4 : 5 : 6 : 7 : ⊥
```

Intermediate Results

Lemma 6.3.4.16 (Lists, Part. Lists, Streams Domain)

$(S_{(L,PL,St)}, \sqsubseteq)$ is a domain with the undefined list $\perp$ as its least element.

Lemma 6.3.4.17 (LPL-Approximants Chain)

The set *LPL-Approx*(`xs`) of a defined list or a defined stream `xs` is a chain.

Theorem 6.3.4.18 (Approximation)

A defined list or a defined stream `xs` is equal to the least upper bound of its LPL-approximants set, also called its limit:

$$\bigsqcup \mathit{LPL\text{-}Approx}(\mathtt{xs}) \;=\; \bigsqcup_{\mathtt{n}=0}^{\infty} \mathit{approx}\ \mathtt{n}\ \mathtt{xs} \;=\; \mathtt{xs}$$

Proof Sketch of Theorem 6.3.4.18 for Lists

Let $xs \equiv (x_0 : x_1 : x_2 : \ldots : x_{len(xs)-1} : [\,])$ be a defined list.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\bigsqcup_{n=0}^{\infty} \mathit{approx}\ n\ xs} \\
= & \bigsqcup\{\bot, & (n = 0) \\
 & \quad x_0 : \bot, & (n = 1) \\
 & \quad x_0 : x_1 : \bot, & (n = 2) \\
 & \quad \ldots & \\
 & \quad x_0 : x_1 : \ldots : x_{n-1} : \bot, & (n = len(xs)) \\
 & \quad x_0 : x_1 : \ldots : x_{n-1} : [\,], & (n = len(xs)+1) \\
 & \quad x_0 : x_1 : \ldots : x_{n-1} : [\,], & (n = len(xs)+2) \\
 & \quad \ldots & \\
 & \quad \} & \\
= & x_0 : x_1 : x_2 : \ldots : x_{n-1} : [\,] & \\
= & x_0 : x_1 : x_2 : \ldots : x_{len(xs)-1} : [\,] & \\
\equiv & xs &
\end{array}
$$

Proof Sketch of Theorem 6.3.4.18 for Streams

Let $xs \equiv (x_0 : x_1 : x_2 : \ldots : x_n : \ldots)$ be a defined stream.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\bigsqcup_{n=0}^{\infty} \mathit{approx}\ n\ xs} \\
= & \bigsqcup \{ \bot, & (n = 0) \\
& \quad x_0 : \bot, & (n = 1) \\
& \quad x_0 : x_1 : \bot, & (n = 2) \\
& \quad \ldots & \\
& \quad x_0 : x_1 : \ldots : x_{m-1} : \bot, & (n = m) \\
& \quad x_0 : x_1 : \ldots : x_m : \bot, & (n = m+1) \\
& \quad x_0 : x_1 : \ldots : x_{m+1} : \bot, & (n = m+2) \\
& \quad \ldots & \\
& \} & \\
= & x_0 : x_1 : x_2 : \ldots : x_n : \ldots & \\
\equiv & xs &
\end{array}
$$

Inductive Proofs on LPL-Approximants Sets

...for proving 'admissible' properties of streams.

Let P be an equational property defined on LPL-approximants and streams.

Proof pattern for defined LPL-approximants:

- Base case: Prove that $P(\perp)$ and $P(\texttt{[]})$ are true.
- Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\texttt{x:xs})$ is true (induction step).

Proof pattern for LPL-approximants w/ possibly undef. values:

- Base case: Prove that $P(\perp)$ and $P(\texttt{[]})$ are true.
- Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\perp\texttt{:xs})$ and $P(\texttt{x:xs})$, $\texttt{x}$ a defined value, are true (induction step).

Homework

Which of the statements of the lemmas in Chapter 6.3.2, 6.3.3, and 6.3.4 hold for

- defined LPL-approximants?
- LPL-approximants with possibly undefined values?

Prove your claims or provide counter-examples.

Note

...the careful distinction between defined and undefined values, between finite lists and finite partial lists, and infinite streams needs to be done analogously for every

- inductively defined Haskell data type

such as trees e.g. (cf. Chapter 6.3.1). Lists, partial lists, and streams just happen to be three most important representatives of inductively defined data structures.

Doing this results in corresponding induction principles for other inductively defined Haskell data types tailored for defined and partial, for finite and infinite values with and without possibly undefined values, etc.

Chapter 6.4

Proof by Approximation

Approximation

...a useful principle for proving equality of infinite objects such as streams, which exploits a conclusion on

- Approximation Theorem 6.3.4.10 and 6.3.4.18

making thereby the proof of equality amenable to

- natural (or mathematical) induction.

L-Approximants

Definition 6.4.1 (L-Approximants)

The set of L-approximants of a defined list or a defined stream `xs` is defined by $\mathit{Approx}(\texttt{xs}) =_{df} \{\, \texttt{take n xs} \mid \texttt{n} \in \mathbb{N}_0 \,\}$

Note, L-approximants are built from the empty list, not the undefined list; they all have finite length.

Examples:

- $\mathit{L\text{-}Approx}(\texttt{[]}) = \{\, \texttt{[]} \,\}$
- $\mathit{L\text{-}Approx}(\texttt{[1,2,3]}) = \{\, \texttt{[]}, \texttt{1:[]}, \texttt{1:2:[]}, \texttt{1:2:3:[]} \,\}$
- $\mathit{L\text{-}Approx}(\texttt{[1..]}) = \{\, \texttt{[]}, \texttt{1:[]}, \texttt{1:2:[]}, \texttt{1:2:3:[]}, \ldots \,\}$

Finiteness, Infinity of Sequences

...in terms of L-approximant sets.

Definition 6.4.2 (Finite, Infinite Sequences)

A sequence of values `xs` is

1. finite, if its L-approximants set $L\text{–}Approx(\texttt{xs})$ is finite.
2. infinite, if its L-approximant sets $L\text{–}Approx(\texttt{xs})$ is infinite.

Lemma 6.4.3 (Finite, Infinite Sequences)

A sequence of values `xs` is

1. finite, i.e., a list, if

   $\exists\, \texttt{m} \in \mathbb{N}.\ (\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{n} \geq \texttt{m}).\ \texttt{take n xs = take (n+1) xs}$
2. infinite, i.e., a stream, if

   $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{take n xs /= take (n+1) xs}$

Equality of Sequences

...in terms of approximant sets.

Definition 6.4.4 (Equality of Sequences)

Two sequences of values xs and ys are equal, if their L-approximant sets are equal, i.e.,

$$\begin{aligned} \textit{L-Approx}(\texttt{xs}) &= \{\, \texttt{take n xs} \mid \texttt{n} \in \mathbb{N} \,\} \\ &= \{\, \texttt{take n ys} \mid \texttt{n} \in \mathbb{N} \,\} \;=\; \textit{L-Approx}(\texttt{ys}) \end{aligned}$$

Lemma 6.4.5 (Equality of Sequences)

Two sequences of values xs and ys are equal, if all their L-approximants are equal, i.e.,

$\forall \texttt{n} \in \mathbb{N}.\ \texttt{take n xs = take n ys}$

Equality of Sequences, Lists and Streams

Corollary 6.4.6 (Finite Sequences)

A sequence of values `xs` is finite, i.e., a list, if

$\exists\, \mathtt{m} \in \mathbb{N}.\ (\forall\, \mathtt{n} \in \mathbb{N}.\ \mathtt{n} \geq \mathtt{m}).\ \mathtt{take\ m\ xs\ =\ take\ (n+1)\ xs}$

Corollary 6.4.7 (Equality of Lists, Streams)

Two lists or two streams `xs` and `ys` are equal, if

$\forall\, \mathtt{n} \in \mathbb{N}.\ \mathtt{take\ n\ xs\ =\ take\ n\ ys}$

Corollary 6.4.8 (Equality of Streams)

Two streams `xs` and `ys` are equal, if

$\forall\, \mathtt{n} \in \mathbb{N}_0.\ \mathtt{xs!!n\ =\ ys!!n}$

Main Results (1)

...reducing the proof of stream equality to a proof of set equality.

Theorem 6.4.9 (Approximation, Stream Equality)

For defined streams $\mathtt{xs}, \mathtt{ys}$ the following claims are equivalent:

1. $xs = ys$
2. $\mathit{LPL\text{-}Approx}(\mathtt{xs}) = \mathit{LPL\text{-}Approx}(\mathtt{ys})$
3. $\mathit{PL\text{-}Approx}(\mathtt{xs}) = \mathit{PL\text{-}Approx}(\mathtt{ys})$
4. $\mathit{L\text{-}Approx}(\mathtt{xs}) = \mathit{L\text{-}Approx}(\mathtt{ys})$

Main Results (2)

...reducing the proof of stream equality to a proof of an equivalent statement accessible to a proof by natural (or mathematical) induction.

Corollary 6.4.10 (Approximation, Stream Equality)

For defined streams $\texttt{xs}, \texttt{ys}$ the following claims are equivalent:

1. $\texttt{xs} = \texttt{ys}$
2. $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{approx n xs} = \texttt{approx n ys}$
3. $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{take}'\ \texttt{n xs} = \texttt{take}'\ \texttt{n ys}$
4. $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{take n xs} = \texttt{take n ys}$
5. $\forall\, \texttt{n} \in \mathbb{N}_0.\ \texttt{xs!!n} = \texttt{ys!!n}$

Note: Proving along the lines of Corollary 6.4.10(5) is usually more convenient than along the lines of Theorem 6.4.9.

Example: Proof by Approximation

Let

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Consider the two definitions facs_mp and facs_zw:

```
facs_mp = map fac [0..]
facs_zw = 1 : zipWith (*) [1..] facs_zw
```

generating the stream of factorials 1,1,2,6,24,120,720,...

We have:

Lemma 6.4.11

$$\forall\, n \in \mathbb{N}_0.\ \texttt{facs\_mp!!n} = \texttt{facs\_zw!!n}$$

Proof by Lemma 6.4.11 (1)

Base case: Let n=0. Equational reasoning yields the desired equality:

```
                  facs_mp!!n
                =   facs_mp!!0
(Def. facs_mp)  =   (map fac [0..])!!0
(L. 6.4.12(1))  =   fac ([0..]!!0)
                =   fac 0
(Def. fac)      =   1
(Def. (!!))     =   (1 : zipWith (*) [1..] facs_zw)!!0
(Def. facs_zw)  =   facs_zw!!0
                =   facs_zw!!n
```

Proof by Lemma 6.4.11 (2)

Inductive case: Let n $\in \mathbb{N}_0$. By means of the induction hypothesis (IH), we can assume `facs_mp!!n` = `facs_zw!!n`. As desired we get:

```
                               facs_mp!!(n+1)
(Def. facs_mp)            =    (map fac [0..])!!(n+1)
(L. 6.4.12(1))            =    fac ([0..]!!(n+1))
(Def. [0..], (!!))        =    fac (n+1)
(Def. fac)                =    (n+1) * fac n
(L. 6.4.12(3))            =    (n+1) * (facs_mp!!n)
(IH)                      =    (n+1) * (facs_zw!!n)
(Def. (!!))               =    ([1..]!!n) * (facs_zw!!n)
(Def. (*))                =    (*) ([1..]!!n) (facs_zw!!n)
(L. 6.4.12(2))            =    (zipWith (*) [1..] facs_zw)!!n
(Def. (!!))               =    (1 : zipWith (*) [1..] facs_zw)!!(n+1)
(Def. facs_zw)            =    facs_zw!!(n+1)
```

Supporting Statement

Lemma 6.4.12

For all natural numbers $n \in \mathbb{N}_0$, we have:

1. `(map f xs)!!n = f (xs!!n)`
2. `(zipWith g xs ys)!!n = g (xs!!n) (ys!!n)`
3. `fac n = facs_mp!!n`

Homework: Prove Lemma 6.4.12.

Homework (1)

Consider the two definitions fibs_memo and fibs_zw:

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

fibs_zw = 0 : 1 : zipWith (+) fibs_zw (tail fibs_zw)
```

generating the stream of Fibonacci numbers
0,1,1,2,3,5,8,13,21,34,55,89,...

Prove by means of natural (mathematical) induction:

Lemma 6.4.13

$$\forall\, n \in \mathbb{N}_0.\ \texttt{fibs\_memo!!n} = \texttt{fibs\_zw!!n}$$

Homework (2)

Why can't we build an inductive proof principle for streams on only L-approximants (cf. Definition 6.4.1)? Compared to the inductive proof principles of Chapter 6.3.4, this would effectively mean to drop or replace the proof of $P(\perp)$ by a proof of $P(\,[]\,)$ in the base case of the inductive proof patterns based on PL- and LPL-approximants of Chapter 6.3.4. Think e.g. on the consequences of 'proving' a property like 'the reverse of the reverse of a stream is the stream itself.'

Chapter 6.4: Further Reading

- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 10, Corecursion – Proof by Approximation)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 17.9, Proof revisited)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17.9, Proof revisited)

Chapter 6.5

Coinduction

Proof by Coinduction

...another useful principle for proving equality of infinite objects such as streams which complements the principle of

- proof by approximation.

Equality as Same Observational Behaviour

...informally, equality of two infinite objects such as streams means that the two objects have

- the same 'observational behaviour.'

For streams, this informally boils down to

- the heads of the streams are the same.
- their tails have the same 'observational behaviour.'

Equality of Streams

...formally, let $[A]$ denote the set of streams over a set of elements A, and let streams $f, g \in [A]$ be written as $f = [f_0, f_1, f_2, f_3, f_4, f_5, \ldots]$ and $g = [g_0, g_1, g_2, g_3, g_4, g_5, \ldots]$.

Definition 6.5.1 (Equality of Streams)

f, $g \in [A]$ are equal iff $\forall\, i \in \mathbb{N}_0.\ f_i = g_i$, i.e., f and g have the same 'observational behaviour.'

...in accordance with Corollary 6.4.8.

In the following

...we will show how to reduce equality of streams to bisimilarity of streams.

This requires the notions of

- Labelled transition systems (LTS) representing streams.
- Stream bisimulation relations capturing the notion of 'same' behaviour of streams.

and some related supporting notions such as

- Expansions of LTS states.
- Bisimilar states.

Labeled Transition Systems

Definition 6.5.2 (Labeled Transition System)

A labeled transition system (LTS) is a tripel (Q, A, T) where

- Q is a set of states.
- A is a set of action labels.
- $T \subseteq Q \times A \times Q$ is a ternary relation, the so-called transition relation.

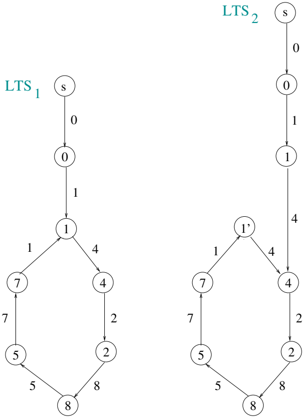
Note: If $(q, a, p) \in T$, we write more conveniently $q \xrightarrow{a} p$.

Example: Representing Streams as LTSs

The decimal representation of $\frac{1}{7}$ has numerous representations as streams of digits, e.g.:

- $0.\overline{142857}$, $0.1\overline{428571}$, $0.14\overline{285714}$, $0.142857142\overline{857142}$,...

LTS_1, LTS_2 are LTS representations of the snd and thd one:



Expansion of LTS States

Let (Q, A, T) be an LTS, and $q \in Q$.

Definition 6.5.3 (Expansion of an LTS State)

1. A finite expansion of q is a finite sequence of actions $[a_0, a_1, a_2, a_3, ..., a_n]$ such that

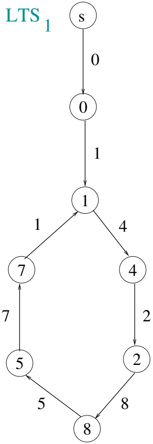
$$(\forall\, i \in \mathbb{N}_0.\ i \leq n).\ \exists\, q_i, q_{i+1} \in Q.\ q_0 = q \wedge q_i \xrightarrow{a_i} q_{i+1}.$$

2. An infinite expansion of q is an infinite sequence of actions $[a_0, a_1, a_2, a_3, ...]$ such that

$$\forall\, i \in \mathbb{N}_0.\ \exists\, q_i, q_{i+1} \in Q.\ q_0 = q \wedge q_i \xrightarrow{a_i} q_{i+1}.$$

Example: Expansion of Digit Stream States

Consider LTS_1 representing digit stream $0.1\overline{428571}$:



The unique infinite expansion of state (i.e., node)

- s is $01\overline{428571}$, 0 is $1\overline{428571}$, 1 is $\overline{428571}$, 2 is $\overline{857142}$,...

Bisimulation Relations, Bisimilar States

Let (Q, A, T) be an LTS, let $p, q \in Q$.

Definition 6.5.4 ((Greatest) Bisimulation Relation)

A bisimulation on (Q, A, T) is a binary relation R on Q, which satisfies: If $q\,R\,p$ and $a \in A$ then:

- $q \xrightarrow{a} q' \;\Rightarrow\; \exists\, p' \in Q.\; p \xrightarrow{a} p' \,\wedge\, q'\,R\,p'$
- $p \xrightarrow{a} p' \;\Rightarrow\; \exists\, q' \in Q.\; q \xrightarrow{a} q' \,\wedge\, q'\,R\,p'$

The largest bisimulation on Q (wrt $\subseteq$) is denoted by $\sim$.

Definition 6.5.5 (Bisimilar States)

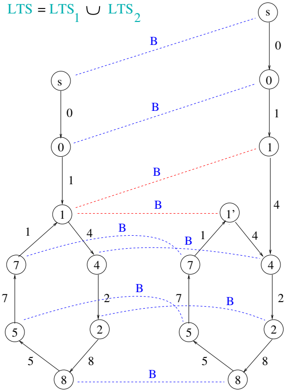
p and q are called bisimilar, if there is a bisimulation R on Q with $q\,R\,p$.

Example: A Bisimulation for Digit Streams

Consider $LTS = (Q, A, T)$ defined as union of LTS_1, LTS_2.

We define relation B on Q as follows:

$\forall\, q, q' \in Q.\ q\, B\, q'$ iff q, q' have the same 'infinite expansion'



$B = \sim$ is the largest bisimulation on the state set Q of LTS.

Streams as Labeled Transition Systems

We introduce the following notation:

If $f = [f_0, f_1, f_2, f_3, f_4, \ldots] \in [A]$ is a stream, then

- f_0 denotes the head
- $\bar{f}$ denotes the tail

of f, i.e., $f = f_0 : \bar{f}$.

Using this notation, f is represented by the below labeled transition system (which unfolds f partially):

$$f \;\bullet \xrightarrow{\quad f_0 \quad} \bullet\; \bar{f}$$

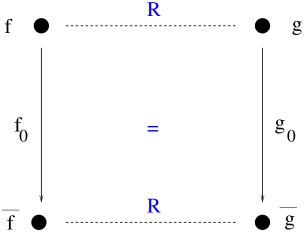
LTS representation of f

Stream Bisimulation

Definition 6.5.6 (Stream Bisimulation)

A stream bisimulation on $[A]$ is a binary relation R on the set of streams $[A]$, which satisfies:

$$\forall\, f, g \in [A].\ f\, R\, g \ \Rightarrow\ f_0 = g_0 \ \wedge\ \bar{f}\, R\, \bar{g}$$



Let $\sim$ denote the largest stream bisimulation on $[A]$.

Reducing Stream Equality

...to largest stream bisimulation.

Let $f = [f_0, f_1, f_2, f_3, f_4, \ldots]$, $g = [g_0, g_1, g_2, g_3, g_4, \ldots] \in [A]$ be two streams with

$$f \xrightarrow{f_0} \bar{f}, \quad g \xrightarrow{g_0} \bar{g}.$$

Then:

Theorem 6.5.7 (Stream Equality as Stream Bisimul.)

f and g are equal iff $f \sim g$, i.e., $f_0 = g_0$ and $\bar{f} \sim \bar{g}$.

Reducing Stream Equality

...further to stream bisimulation.

By definition, $\sim$ is the largest stream bisimulation. This yields:

Lemma 6.5.8

$$f \sim g \;\Leftrightarrow\; \exists\, B.\ B \text{ stream bisimulation on } [A] \;\wedge\; f\, B\, g$$

Together, Theorem 6.5.7 and Lemma 6.5.8 imply:

Corollary 6.5.9

f and g are equal iff

$$\exists\, B.\ B \text{ stream bisimulation on } [A] \;\wedge\; f\, B\, g$$

Coinductive Proof Pattern

...using Corollary 6.5.9, proving the equality of two streams f and g of $[A]$ requires:

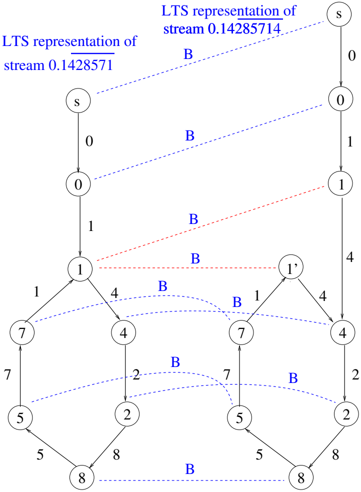
1. Finding a relation B on $[A]$.
2. Proving that B is a stream bisimulation with $f\ B\ g$.

...considering Haskell streams, this means proving the equality of two Haskell streams `xs` and `ys` requires:

1. Finding a relation B on the set of Haskell streams.
2. Proving that B is a stream bisimulation with `xs` B `ys`.

Example: Stream Bisimulation $B \subseteq \sim$

...for streams $0.1\overline{428571}$ and $0.14\overline{285714}$:



...$0.1\overline{428571}$, $0.14\overline{285714}$ are stream bisimilar and hence equal.

Chapter 6.5: Further Reading (1)

- Falk Bartels. *Generalized Coinduction*. Electronic Notes in Theoretical Computer Science 44:1, 21 pages, 2001. http://www.elsevier.nl/locate/entcs/volume44.html
- Falk Bartels. *Generalized Coinduction*. Journal of Mathematical Structures in Computer Science 13(2):321-348, 2003.
- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 10.3, Proof by Approximation; Chapter 10.4, Proof by Coinduction)
- Chung-Kil Hur, Georg Neis, Derek Dreyer, Viktor Vafeiadis. *The Power of Parameterization in Coinductive Proofs*. In Conference Record of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 193-205, 2013.

Chapter 6.5: Further Reading (2)

- Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62:222-259, 1997.
- Marina Lenisa. *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*. Electronic Notes in Theoretical Computer Science 19:2-22, 1999.
- Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Appendix B.2, Introducing Coinduction; Appendix B.3, Proof by Coinduction)

Chapter 6.5: Further Reading (3)

- Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. Theoretical Computer Science 308:1-53, 2003.
- Davide Sangiorgi. *On the Bisimulation Proof Method*. Journal of Mathematical Structures in Computer Science 8:447-479, 1998.
- Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
- Davide Sangiorgi, Jan Rutten (Eds). *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Vol. 52, Cambridge University Press, 2011.

Chapter 6.6

Fixed Point Induction

Fixed Point Induction

...a useful proof principle allowing us to prove properties of the

- least fixed point of continuous functions

on complete partial orders or more specifically complete lattices, which are both specific partially ordered sets (refer to Appendix A for definitions of terms, if required).

Admissible Predicates

Let $(C, \sqsubseteq)$ be a complete partial order (CPO) (or domain), and ψ be a predicate on C, i.e., $\psi : C \to IB$.

Definition 6.6.1 (Admissible Predicate)

ψ is called admissible iff for every chain $D \subseteq C$ holds:

$$(\forall\, d \in D.\ \psi(d)) \;\Rightarrow\; \psi(\bigsqcup D)$$

Lemma 6.6.2

ψ is admissible, if it is expressible as an equation.

Example: Streams, Sequences of Approximants

Recalling that $(S_{(PL,St)}, \sqsubseteq)$ with $S_{(PL,St)}$ the set of streams and partial lists (cf. Definition 6.3.4.3), and $\sqsubseteq$ the approximation order defined in Lemma 6.3.4.4, is a CPO (or domain) (cf. Lemma 6.3.4.8), we get as corollary:

Corollary 6.6.3

Let ψ be a predicate on the set of partial lists and streams $S_{(PL,St)}$ expressible as an equation, let s be a stream, and $S' \subseteq S$ the infinite chain of its PL-approximants (cf. Definition 6.3.4.5) with $\bigsqcup S' = s$. Then:

$$(\forall s' \in S'.\ \psi(s')) \Rightarrow \psi(\bigsqcup S') \quad (\Leftrightarrow \psi(s))$$

Monotonic and Continuous Functions on CPOs

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be CPOs, and let $f \in [C \to D]$ be a map from C to D.

Definition 6.6.4 (Monotonic, Continuous Maps)

f is called

1. monotonic (or order preserving) iff
$$\forall\, c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$$
(Preservation of the ordering of elements)
2. continuous iff f is monotonic and
$$(\forall\, C' \subseteq C.\ C' \neq \emptyset \wedge C' \ \mathit{chain}).\ f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$$
(Preservation of least upper bounds)

Fixed Points, Least Fixed Points

...of continuous functions on complete partial orders (CPOs).

Definition 6.6.5 (Fixed Point, Least Fixed Point)

Let $(C, \sqsubseteq)$ be a complete partial order, let $f \in [C \stackrel{con}{\to} C]$ be a continuous function on C, and let $c \in C$ be an element of C. Then:

1. c is called a fixed point of f iff $f(c) = c$.
2. c is called the least fixed point of f, denoted by μf, iff $\forall\, d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$

Note: Fixed Point Theorem A.5.1.3 of Knaster, Tarski, and Kleene ensures the existence of least fixed points of continuous functions on CPOs.

Fixed Point Induction

...the general pattern of fixed point induction:

Theorem 6.6.6 (Fixed Point Induction)

Let $(C, \sqsubseteq)$ be a complete partial order (CPO), let $f : C \to C$ be a continuous function on C, and let $\psi : C \to IB$ be an admissible predicate on C. Then:

$$(\forall\, c \in C.\ \psi(c) \;\Rightarrow\; \psi(f(c))) \;\Rightarrow\; \psi(\mu f)$$

where μf denotes the least fixed point of f.

Proof Sketch of Theorem 6.6.6

- The empty set $\emptyset \subseteq C$ is (trivially) a chain.
- Since C is a CPO, $\bigsqcup \emptyset$ *exists* $= \perp_C$ with $\perp_C$ the least element of C.
- ψ admissible yields $\psi(\perp_C)$.
  (Note that $(\forall\, d \in \emptyset.\ \psi(d))$ holds trivally; ψ admissible thus implies $\psi(\bigsqcup \emptyset) = \psi(\perp_C) = true$.)
- Using the assumptions of Theorem 6.6.6, we can prove by induction on $n \in \mathbb{N}_0$:
  - $D =_{df} \{f^n(\perp_C) \mid n \in \mathbb{N}_0\} \subseteq C$ is a chain.
  - $\forall\, n \in \mathbb{N}_0.\ \psi(f^n(\perp_C))$.
- D chain, $\forall\, d \in D.\ \psi(d)$, ψ admissible, yields $\psi(\bigsqcup D)$.
- Last but not least, Fixed Point Theorem A.5.1.3 (Knaster, Tarski, Kleene) yields $\mu f = \bigsqcup D$.
- Thus, we obtain $\psi(\mu f)$, which completes the proof.

Chapter 6.6: Further Reading

- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 6, Axiomatic Program Verification – Fixed Point Induction)
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 9, Axiomatic Program Verification – Fixed Point Induction)

Chapter 6.7

Other Approaches, Verification Tools

Chapter 6.7.1

Correctness by Construction

Correctness by Construction

...conceptually, testing and verification are

- *a posteriori* approaches

for proving correctness of a program as they are applied after the program development is finished.

Conceptually dual to testing and verification is the approach of

- correctness by construction

which strives to prove correctness of a program on the fly of its development by proving correctness of every step of the development.

Hence, correctness by construction is conceptually an

- *a priori* (or *on-the-fly*) approach.

Techniques for Correctness by Correctness

...in principle, every proof technique can be made use of by approaches aiming at correctness by construction.

This includes the inductive proof principles discussed in Chapter 6 as well as equational reasoning discussed in Chapter 4, which sometimes is also called proof by program calculation.

Approaches for proven correct rule-based program transformations, however, are prevailing and thus of particular importance.

Functional Pearls: Correctness by Construction

...the development of a functional pearl starting with a program being

- obviously correct (but inefficient)

by a sequence of transformation steps into a program being (more)

- efficient and still correct

since (ideally) every transformation step is proved correct (cp. Chapter 4), can be considered an approach in the spirit of ensuring correctness by construction.

Chapter 6.7.1: Further Reading (1)

- Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
- Richard Bird. *How to Write a Functional Pearl*. Invited presentation at the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 2006. http://icfp06.cs.uchicago.edu/bird-talk.pdf
- Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011.
- Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.

Chapter 6.7.1: Further Reading (2)

- Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98). Springer-V., LNCS 1422, 122-150, 1998.
- Anthony Hall, Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. IEEE Software 19(1):18-25, 2002.
- Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.
- Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.

Chapter 6.7.2

Selected other Approaches and Tools

Other Approaches and Tools: A Selection (1)

- Programming by contracts (Vytiniotis et al., POPL 2013)
- Verifying equational properties of functional programs (Sonnex et al., TACAS 2012)
  - Tool Zeno: Proof search is based on induction and equality reasoning which are driven by syntactic heuristics.
- Verifying first-order and call-by-value recursive functional programs (Suter et al., SAS 2011)
  - Tool Leon: Based on extending SMT to recursive programs.

Other Approaches and Tools: A Selection (2)

- Verifying higher-order functional programs (Unno et al., POPL 2013)
  - Tool MoCHi-X: Prototype implementation of a type inference algorithm as extension of the software model checker MoChi (Kobayashi et al., PLDI 2011).
- Verifying lazy Haskell (Mitchell et al., Haskell 2008)
  - Tool Catch: Based on static analysis; can prove absence of pattern matching failures; evaluated on 'real' programs.
- ...

Verified Functional Programming in Agda

- Aaron Stump. Verified Functional Programming in Agda. ACM Books Series, No. 9, 2016.

...a text snippet from the book:

'Agda is an advanced programming language based on Type Theory. Agda's type system is expressive enough to support full functional verification of programs, in two styles.

In external verification, we write pure functional programs and then write proofs of properties about them. The proofs are separate external artifacts, typically using structural induction.

In internal verification, we specify properties of programs through rich types for the programs themselves. This often necessitates including proofs inside code, to show the type checker that the specified properties hold.

The power to prove properties of programs in these two styles is a profound addition to the practice of programming, giving programmers the power to guarantee the absence of bugs, and thus improve the quality of software more than previously possible.'

Chapter 6.8

References, Further Reading

Chapter 6: Further Reading (1)

- Martin Aigner, Günter M. Ziegler. *Proofs from the Book*. Springer-V., 4th edition, 2010.
- André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
- Roland Backhouse, Roy Crole, Jerimy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002.
- Falk Bartels. *Generalized Coinduction*. Electronic Notes in Theoretical Computer Science 44:1, 21 pages, 2001. http://www.elsevier.nl/locate/entcs/volume44.html
- Falk Bartels. *Generalized Coinduction*. Journal of Mathematical Structures in Computer Science 13(2):321-348, 2003.

Chapter 6: Further Reading (2)

- Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
- Richard Bird. *How to Write a Functional Pearl*. Invited presentation at the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 2006. http://icfp06.cs.uchicago.edu/bird-talk.pdf
- Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011.
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 6, Proofs)
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 5, Induction and Recursion)

Chapter 6: Further Reading (3)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18, Programme verifizieren und testen)

- Matthias Blume, David McAllester. *Sound and Complete Models of Contracts*. Journal of Functional Programming 16(4-5):375-414, 2006.

- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 9.1.2, Induktion; Kapitel 9.1.3, Strukturelle Induktion; Kapitel 9.2, Mit Lemmata und Generalisierung arbeiten; Kapitel 9.3, Programmherleitung)

Chapter 6: Further Reading (4)

- Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.
- Werner Damm, Bernhard Josko. *A Sound and Relatively* Complete Hoare-Logic for a Language with Higher Type Procedures*. Acta Informatica 20:59-101, 1983.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 9, Correctness)
- Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98). Springer-V., LNCS 1422, 122-150, 1998.

Chapter 6: Further Reading (5)

- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 3, The Use of Logic: Proof – Proof Style, Proof Recipes, Strategic (Proof) Guidelines; Chapter 7, Induction and Recursion; Chapter 10, Corecursion – Proof by Approximation, Proof by Coinduction; Chapter 11.1, More on Mathematical Induction)
- Andreas Goerdt. *A Hoare Calculus for Functions defined by Recursion on Higher Types*. In Proceedings of the Conference on Logic of Programs, Springer-V, LNCS 193, 106-117, 1985.
- Anthony Hall, Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. IEEE Software 19(1):18-25, 2002.

Chapter 6: Further Reading (6)

- Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 11, Proof by Induction; Chapter 14.6, Inductive Properties of Infinite Lists)
- Chung-Kil Hur, Georg Neis, Derek Dreyer, Viktor Vafeiadis. *The Power of Parameterization in Coinductive Proofs*. In Conference Record of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 193-205, 2013.
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016. (Chapter 16, Reasoning about programs)

Chapter 6: Further Reading (7)

- Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62:222-259, 1997.
- Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko. *HMC: Verifying Functional Programs using Abstract Interpreters*. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), Springer-V., LNCS 6806, 470-485, 2011.
- Steve King, Jonathan Hammond, Roderick Chapman, Andy Pryor. *Is Proof More Cost-Effective than Testing?* IEEE Transactions on Software Engineering 26(8):675-686, 2000.

Chapter 6: Further Reading (8)

- Naoki Kobayashi, Ryosuke Sato, Hiroshi Unno. *Predicate Abstraction and CEGAR for Higher-Order Model Checking*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), 222-233, 2011.
- Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.
- Marina Lenisa. *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*. Electronic Notes in Theoretical Computer Science 19:2-22, 1999.

Chapter 6: Further Reading (9)

- David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 4, Recycling Outputs as Inputs: Induction and Recursion; Chapter 4.1, What are Induction and Recursion? Chapter 4.6, Structural Recursion and Induction; Chapter 4.7, Recursion and Induction on Well-Founded Sets)
- Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
- Neil Mitchell, Colin Runciman. *Not all Patterns, but enough: An Automated Verifier for Partial but Succifient Pattern Matching*. In Proceedings of the 1st ACM SIG-PLAN Symposium on Haskell (Haskell 2008), 49-60, 2008.

Chapter 6: Further Reading (10)

- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Appendix B, Induction and Coinduction)
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. Chapter 1, Introduction – Structural Induction; Chapter 6, Axiomatic Program Verification – Fixed Point Induction)
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. Chapter 1, Introduction – Structural Induction; Chapter 9, Axiomatic Program Verification – Fixed Point Induction)

Chapter 6: Further Reading (11)

- Lawrence C. Paulson. *Logic and Computation – Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987. (Chapter 4, Structural Induction; Chapter 10, Sample Proofs (with Cambridge LCF))
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 11.4.1, Über wohlfundierte Ordnungen; Kapitel 11.4.2, Wie beweist man Terminierung?)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten)
- Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. Theoretical Computer Science 308:1-53, 2003.

- Davide Sangiorgi. *On the Bisimulation Proof Method*. Journal of Mathematical Structures in Computer Science 8:447-479, 1998.
- Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
- Davide Sangiorgi, Jan Rutten (Eds). *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Vol. 52, Cambridge University Press, 2011.
- William Sonnex, Sophia Drossopoulou, Susan Eisenbach. *Zeno: An Automated Prover for Properties of Recursive Data Structures*. In Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012), Springer-V., LNCS 7214, 407-421, 2012.

Chapter 6: Further Reading (13)

- Aaron Stump. *Verified Functional Programming in Agda*. ACM Books Series, No. 9, 2016.
- Philippe Suter, Ali Sinan Köksal, Viktor Kuncak. *Satisfiability Modulo Recursive Programs*. In Proceedings of the 18th International Conference on Static Analysis (SAS 2011), Springer-V., LNCS 6887, 298-315, 2011.
- Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik: Induktives Vorgehen*. Springer-V., 2014. (Chapter 4, Induktives Definieren; Chapter 5, Induktives Beweisen; Chapter 6, Induktives Vorgehen: Potential und Grenzen)

Chapter 6: Further Reading (14)

- Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Chapter 4, Inductive Definitions; Chapter 5, Inductive Proofs; Chapter 6, Inductive Approach: Potential, Limitations, and Pragmatics)
- Simon Thompson. *Proof*. In *Research Directions in Parallel Functional Programming*, Kevin Hammond, Greg Michaelson (Eds.), Springer-V., Chapter 4, 93-119, 1999.
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 8, Reasoning about programs; Chapter 17.9, Proof revisited)

Chapter 6: Further Reading (15)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about programs; Chapter 17.9, Proof revisited)
- Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008. (Chapter 105, Software Testing; Chapter 106, Formal Methods; Chapter 107, Verification and Validation)
- Hiroshi Unno, Tachio Terauchi, Naoki Kobayashi. *Automating Relatively Complete Verification of Higher-Order Functional Programs*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 75-86, 2013.

Chapter 6: Further Reading (16)

- Daniel J. Velleman. *How to Prove It. A Structured Approach*. Cambridge University Press, 1994.
- Dimitrios Vytiniotis, Simon Peyton Jones, Dan Rosén, Koen Claessen. *HALO: Haskell to Logic through Denotational Semantics*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 431-442, 2013.
- Mitchell Wand. *Induction, Recursion, and Programming*. Elsevier, 1980.

Part IV

Advanced Language Concepts

Chapter 7

Functional Arrays

Chapter 7.1

Motivation

Distinctive

...properties of imperative arrays:

+ Values of an array can be accessed or updated in constant time.
+ The update operation does not need extra space.
+ There is no need for chaining the array elements with pointers as they can be stored in contiguous memory locations.

– Their size is fixed (defined at the time of declaration).

Functional Lists and Arrays

Functional lists

- – do not enjoy the set of favorable properties of imperative arrays; most importantly, values of a list can not be accessed or updated in constant time.
  - Accessing the ith element of a list (using `(!!)`) takes a number of steps proportional to i.
- + can be arbitrarily long, potentially even infinite.

Functional arrays

- + are designed and implemented to get as close as possible to the properties of imperative arrays.
  - Accessing the ith element of an array (using `(!)`) takes a constant number of steps, regardless of i.
- – are of fixed size (defined at the time they are created).

Functional Arrays

...are not supported by the standard prelude of Haskell but by various libraries

- `import Array`
- `import Data.Array.IArray`
- `import Data.Array.Diff`

providing different kinds and implementations of functional arrays:

- Static arrays (w/out destructive update)
- Dynamic arrays (w/ destructive update)

Chapter 7.2

Functional Arrays

Chapter 7.2.1

Static Arrays

Static Arrays

...are supported by the library `Array`:

- `import Array`

which provides three functions for creating static arrays:

- `array` *bounds list_of_associations*
- `listArray` *bounds list_of_values*
- `accumArray` *f init bounds list_of_associations*

In more detail

...the three functions for creating static arrays:

- `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 Index Type Class Ix

...extends the type class `Ord` (and indirectly 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
```

Members of `Ix`

- must provide implementations of `range`, `index`, `inRange`, and `rangeSize`.
- are (mainly) used for indices of arrays.

Creating Static Arrays: 1st Mechanism

...using the function `array`, the most fundamental means:

- `array :: Ix a => (a,a) -> [(a,b)] -> Array a b`
  `array` *bounds* *list_of_associations*

where

- *bounds* specifies the values of the smallest and array largest index.

  Example: The bound values `(0,4)` and `((1,1),(10,10))` specify a
  - zero-origin vector of length five
  - one-origin 10 by 10 matrix, respectively.

  Note: The components of *bounds* can be given by arbitrary expressions.
- *list_of_associations* is a list of associations of the form `(i,x)` specifying that the value of the array element at index position `i` is `x`.

Examples

Let a′, `f n`, and `m` be the following expressions:

```
a'  = array (1,4) [(3,'c'),(2,'a'),(1,'f'),(4,'e')]
f n = array (0,n) [(i,i*i) | i <- [0..n]]
m   = array ((1,1),(2,3))
            [((i,j),(i*j)) | i <- [1..2], j <- [1..3]]
```

These expressions have type

```
a' :: Array Int Char
f  :: Int -> Array Int Int
m  :: Array (Int,Int) Int
```

and value

```
a'  ->> array (1,4) [(1,'f'),(2,'a'),(3,'c'),(4,'e')]
f 3 ->> array (0,3) [(0,0),(1,1),(2,4),(3,9)]
m   ->> array ((1,1),(2,3)) [((1,1),1),((1,2),2),
                             ((1,3),3),((2,1),2),
                             ((2,2),4),((2,3),6)]
```

Remarks

...arrays have type `Array a b` where

- `a` represents the type of the index
- `b` represents the type of the values of array elements.

Note:

- An array is undefined if any specified index is out of bounds.
- If two associations in the association list have the same index, the value at that index is undefined.

  This means: The function `array` is strict in bounds but non-strict (lazy) in values. Arrays can thus contain 'undefined' elements.

Examples

Computing Fibonacci numbers:

```
fibs n = a
 where a = array (1,n) ([(1,0), (2,1)] ++
                        [(i, a!(i-1) + a!(i-2))
                          | i <- [3..n]])
```

Applications:

```
fibs  3 ->> array (1,3) [(1,0),(2,1),(3,1)]
fibs  5 ->> array (1,5) [(1,0),(2,1),(3,1),
                         (4,2),(5,3)]
fibs 12 ->> array (1,12) [(1,0),(2,1),(3,1),
                          (4,2),(5,3),(6,5),
                          (7,8),(8,13),(9,21),
                          (10,34),(11,55),(12,89)]
```

The Array Access Function (!)

...the array access function `(!)`

```
(!) :: Ix a => Array a b -> a -> b
```

returns the value `v :: b` at index position `i :: a`.

Recall: The index type must be a member of type class `Ix`, which provides maps specifically needed for index operations.

Examples

Computing Fibonacci numbers:

```
fibs n = a
 where a = array (1,n) ([(1,0), (2,1)] ++
                         [(i, a!(i-1) + a!(i-2))
                           | i <- [3..n]])
```

Applications of (!):

```
fibs 5!5      ->> 3
fibs 10!10    ->> 34
fibs 100!10   ->> 34 -- Thanks to lazy evaluation
                     -- computation stops at
                     -- fibs 10!10
fibs 50!50    ->> 7.778.742.049
fibs 100!100 ->> 218.922.995.834.555.169.026
fibs 5!10     ->> Program error: Ix.index: index
                  out of range
```

A Note on Performance

Declaring `a` locally in a `where`-clause in the definition of `fibs`

- avoids creating new arrays during computation
- is crucial for performance.

For comparison consider the definition of `xfibs`, where `a` (of a slightly different type) is globally defined:

```
xfibs n = a n
a n     = array (1,n) ([(1,0),(2,1)] ++
                        [(i,a n!(i-1) + a n!(i-2))
                          | i <- [3..n]])
```

Examples

Applications:

```
xfibs  3 ->> array (1,3) [(1,0),(2,1),(3,1)]
xfibs  5 ->> array (1,5) [(1,0),(2,1),(3,1),(4,2),(5,3)]
xfibs 12 ->> array (1,12) [(1,0),(2,1),(3,1),
                            (4,2),(5,3),(6,5),
                            (7,8),(8,13),(9,21),
                            (10,34),(11,55),(12,89)]

xfibs 5!5   ->> 3
xfibs 10!10 ->> 34
xfibs 25!20 ->> 4.181   -- thanks to lazy evaluation
                        -- the computation stops asap

xfibs 25!25 ->> ...takes too long to be feasible!
```

Note: Though correct, evaluating `xfibs n` is most inefficient due to the creation of new arrays during the evaluation.

Creating Static Arrays: 2nd Mechanism

...using the function `listArray`, a more sophisticated means:

- `listArray :: (Ix a) => (a,a) -> [b] -> Array a b`
  `listArray` *bounds* *list_of_values*

where

- *bounds* specifies the values of the smallest and the largest index.
- *list_of_values* specifies the values of the array elements in terms of a list.

Note: The function `listArray` is especially useful

- for the frequently occurring case where an array is constructed from a list of values given in index order.

Example

```
a'' :: Array Int Char
a'' = listArray (1,8) "fun prog"

a'' ->> array (1,8) [(1,'f'),(2,'u'),(3,'n'),(4,' '),
                     (5,'p'),(6,'r'),(7,'o'),(8,'g')]
```

Creating Static Arrays: 3rd Mechanism

...using the function `accumArray`, the most powerful means:

- ```
  accumArray :: (Ix a) => (b -> c -> b) -> b
                        -> (a,a) -> [(a,c)] -> Array a b
  ```
  `accumArray` *f init bounds list_of_associations*

where

- *f* specifies an accumulation function.
- *init* specifies the (default) value the elements of the array shall be initialized with.
- *bounds* specifies the values of the smallest and the largest index.
- *list_of_associations* specifies the values of the array in terms of an association list.

Note: `accumArray` does not require that the indices occurring in list_of_associations are pairwise disjoint. Instead, values of 'conflicting' indices are accumulated via *f*.
  ```

Example 1: A Histogram Function

...using the function accumArray:

```
histogram :: (Ix a, Num b) =>
                         (a,a) -> [a] -> Array a b
histogram bounds vs =
 accumArray (+) 0 bounds [(i,1) | i <- vs]
```

Applications:

```
histogram (1,5) [4,1,4,3,2,5,5,1,2,1,3,4,2,1,1,3,2,1]
 ->> array (1,5) [(1,6),(2,4),(3,3),(4,3),(5,2)]

histogram (-1,4) [1,3,1,1,3,1,1,3,1]
 ->> array (-1,4) [(-1,0),(0,0),(1,6),(2,0),(3,3),(4,0)]

histogram (1,3) [5,3,1,3,4,2,(-4),1,1,3,2,1,5,(-9)]
 ->> array
     Program error: Ix.index: index out of range
```

Example 2: A Prime Number Test

...using the function accumArray:

```
primes :: Int -> Array Int Bool
primes n =
   accumArray (\e e' -> False) True (2,n) l
    where l = concat [map (flip (,) ())
                     (takeWhile (<=n) [k*i|k<-[2..]])
                                   | i<-[2..n `div` 2]]
```

Applications:

```
(primes 100)!1   ->> Program error: Ix.index: index
                     out of range
(primes 100)!2   ->> True
(primes 100)!4   ->> False
(primes 100)!71  ->> True
(primes 100)!100 ->> False
(primes 100)!101 ->> Program error: Ix.index: index
                     out of range
```

Array Operators (1)

...pre-defined array operators:

- `(!)`: array subscripting, yields the *i*th element of an array.
- `bounds`: yields the smallest and largest index of an array.
- `indices`: yields a list of the indices of an array.
- `elems`: yields a list of the elements/values of an array.
- `assocs`: yields a list of index/value pairs of the elements of an array, i.e., the list of associations of an array.
- `(//)`: array updating – `(//)` takes an array (left argument) and a list of associations (right argument) and returns a new array, which is identical to the argument array except for the values of elements occurring in the argument list of associations.

  Note: `(//)` generates a modified copy of the argument array; it does not perform a destructive update!
- ...

Array Operators (2)

...the syntactic signatures of the array operators:

- `(!)      :: (Ix a) => Array a b -> a -> b`
- `bounds   :: (Ix a) => Array a b -> (a,a)`
- `indices  :: (Ix a) => Array a b -> [a]`
- `elems    :: (Ix a) => Array a b -> [b]`
- `assocs   :: (Ix a) => Array a b -> [(a,b)]`
- ```
  (//)     :: (Ix a) => Array a b -> [(a,b)]
                                    -> Array a b
  ```
- ...
  ```

Example: The Prime Number Test

Applications (w/ pre-defined functions on arrays):

```
elems (primes 10)
 ->> [True,True,False,True,False,True,False,False,False]

assocs (primes 10)
 ->> [(2,True),(3,True),(4,False),(5,True),(6,False),
      (7,True),(8,False),(9,False),(10,False)]

yieldPrimes (assocs (primes 100))
 ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
      59,61,67,71,73,79,83,89,97]
```

where

```
yieldPrimes :: [(a,Bool)] -> [a]
yieldPrimes [] = []
yieldPrimes ((v,w):t)
 | w         = v : yieldPrimes t
 | otherwise = yieldPrimes t
```

Example: More Uses of the Array Operators

The setting:

```
m = array ((1,1),(2,3)) [((i,j),i*j) | i <- [1..2],
                                       j <- [1..3]]
                                 :: Array (Int,Int) Int

m ->> array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
                           ((2,1),2),((2,2),4),((2,3),6)]

m!(1,2) ->> 2, m!(2,2) ->> 4, m!(2,3) ->> 6
```

Applications of array operators:

```
bounds  m ->> ((1,1),(2,3))
indices m ->> [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
elems   m ->> [1,2,3,2,4,6]
assocs  m ->> [((1,1),1),((1,2),2),((1,3),3),
               ((2,1),2), ((2,2),4), ((2,3),6)]

m // [((1,1),4), ((2,2),8)]
 ->> array ((1,1),(2,3)) [((1,1),4),((1,2),2),((1,3),3),
                          ((2,1),2),((2,2),8),((2,3),6)]
```

Updating Arrays: (//) vs. accum

...`accum`, another pre-defined function on arrays:

- `accum :: (Ix a) => (b -> c -> b) -> Array a b`
  `-> [(a,c)] -> Array a b`

  `accum` *f a list_of_associations*

...instead of replacing previously stored values as `(//)` does, `accum` accumulates values referring to the same index using `f`.

Application:

```
accum (+) m [((1,1),4), ((2,2),8)]     -- m as before
 ->> array ((1,1),(2,3))
           [((1,1),5),((1,2),2),((1,3),3),
            ((2,1),2),((,2,2),12),((2,3),6)]
```

Note: The result of `accum` is a new matrix, which is identical to `m` except for the elements at positions `(1,1)` and `(2,2)` to whose values `1` and `4`, `4` and `8` have been added, respectively.

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
Part IV
Chap. 7
7.1
7.2
7.2.1
7.2.2
7.3
7.4
Chap. 8
Chap. 9
Chap. 10

Example: A Modified Histogram Function

...illustrating the update operator (//):

```
histogram (lower,upper) xs
 = updHist (array (lower,upper)
                  [(i,0) | i <- [lower..upper]])
           xs

updHist a []     = a
updHist a (x:xs) = updHist (a // [(x, (a!x + 1))]) xs
```

Application:

```
histogram (0,9) [3,1,4,1,5,9,2]
 ->> array (0,9) [(0,0),(1,2),(2,1),(3,1),(4,1),
                  (5,1),(6,0),(7,0),(8,0),(9,1)]
```

Pre-defined Higher-Order Array Functions

...higher-order functions can be defined on arrays just as on lists.

Examples:

```
amap :: (b -> c) -> Array a b -> Array a c
amap (\x -> x*10) a
```

...yields an array where all elements of a are multiplied by 10.

```
ixmap :: (Ix a, Ix b) => (a,a) -> (a -> b)
                             -> Array b c -> Array a c
ixmap b f a = array b [(k,a!f k) | k <- range b]
```

User-defined Higher-Order Array Functions

The functions `row` and `col` return a row and a column of a matrix, respectively:

```
row :: (Ix a,Ix b) =>
                 a -> Array (a,b) c -> Array b c
row i m = ixmap (l',u') (\j->(i,j)) m
    where ((l,l'),(u,u')) = bounds m

col :: (Ix a,Ix b) =>
                 a -> Array (b,a) c -> Array b c
col j m = ixmap (l,u) (\i->(i,j)) m
    where ((l,l'),(u,u')) = bounds m
```

Example: Uses of `row` and `col`

Applications (with `m` as before):

```
row 1 m ->> array (1,3) [(1,1),(2,2),(3,3)]
row 2 m ->> array (1,3) [(1,2),(2,4),(3,6)]
row 3 m ->> array (1,3) [(1,
               Program error: Ix.index: index out of
               range

col 1 m ->> array (1,2) [(1,1),(2,2)]
col 2 m ->> array (1,2) [(1,2),(2,4)]
col 3 m ->> array (1,2) [(1,3),(2,6)]
col 4 m ->> array (1,2) [(1,
               Program error: Ix.index: index out of
               range
```

Chapter 7.2.2

Dynamic Arrays

Dynamic Arrays

...are supported by the library `Data.Array.Diff`:

- `import Data.Array.Diff`

The type

- `DiffArray` (for dynamic arrays)

replaces the type

- `Array` (for static arrays)

...everything else behaves analogously.

Chapter 7.3

Summary

Summing up

Static Arrays

- Access operator (!): Access to each array element in constant time.
- Update operator (//): No destructive update; instead an identical copy of the argument array is created except of those elements being 'updated.' Updates thus do not take constant time.

Dynamic Arrays

- Update operator (//): Destructive update; updates take constant time per index.
- Access operator (!): Access to array elements may sometimes take longer as for static arrays.

Note

Updates

- can often completely be avoided by smartly written recursive array constructions (cp. the prime number test in Chapter 7.2.1).

Dynamic arrays

- should only be used if constant time updates are crucial for the application.

For an extended example showing

- arrays at work.

refer to Chapter 16.2 dealing with an imperative robot language for controlling robot actions.

Chapter 7.4

References, Further Reading

Chapter 7: Further Reading (1)

- Henry G. Baker. *Shallow Binding Makes Functional Arrays Fast*. ACM SIGPLAN Notices 26(8):145-147, 1991.
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Chapter 10.1, Arrays)
- Manuel M.T. Chakravarty, Gabriele Keller. *An Approach to Fast Arrays in Haskell*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 27-58, 2003.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.6, Arrays)

Chapter 7: Further Reading (2)

- Klaus E. Grue. *Arrays in Pure Functional Programming Languages*. International Journal on Lisp and Symbolic Computation 2:105-113, Kluwer Academic Publishers, 1989.
- Paul Hudak. *Arrays, Non-determinism, Side-effects, and Parallelism: A Functional Perspective*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 312-327, 1986.
- 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)

- John Hughes. *An Efficient Implementation of Purely Functional Arrays*. Technical Report, Programming Methodology Group, Chalmers University of Technology, 1985.
- Melissa E. O'Neill, F. Warren Burton. *A New Method for Functional Arrays*. Journal of Functional Languages 7(5):487-513, 1997.
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 14, Funktionale Arrays und numerische Mathematik)

Chapter 7: Further Reading (4)

- Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. `www.haskell.org/definitions`. (Chapter 16, Arrays)
- Simon Peyton Jones. *Haskell 98 Libraries: Arrays*. Journal of Functional Programming 13(1):173-178, 2003.
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 2.7, Arrays; Chapter 4.3, Arrays)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 12, Barcode Recognition – Introducing Arrays)

Chapter 7: Further Reading (5)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 19, Time and space behaviour – arrays)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 20, Time and space behaviour)
- Philip Wadler. *A New Array Operation*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 328-335, 1986.

Chapter 8

Abstract Data Types

Chapter 8.1

Motivation

Concrete Data Types (CDTs)

...are specified by naming their values (not by naming their operations):

- With the exception of functions as values of a CDT, every CDT value is uniquely described by an expression composed of constructors.
- Using pattern matching, these expressions can be generated, inspected, and modified in various ways by operations associated with the CDT.
- There is no need, however, to specify any operation associated with a CDT at the time of defining it.

...the Haskell means for defining CDTs are algebraic (and new type) data type definitions.

Illustration: CDTs, CDT Values in Haskell

```
type Forename  = String
...
type Publisher = String
type Edition   = Int

data Vehicle = Bicycle | Motorcycle | Car | Bus
data Tree a  = Nil | Leaf a | Root (Tree a) a (Tree a)
data Person  = P Forename Surname Address
newtype Book = B (Author,Title,Publisher,Edition)

v1 = Bicycle :: Vehicle
v2 = Car :: Vehicle
t1 = Leaf 42 :: Tree Int
t2 = Root Nil True (Leaf False) :: Tree Bool
p  = P "Simon" "Thompson" "unknown" :: Person
b  = B ("Thompson","Haskell","Addison-Wesley",2) :: Book
```

Note: At the time of defining the above CDTs, there is no need to define operations manipulating their values.

Abstract Data Types (ADTs)

...are specified by naming their operations (not by naming their values):

- The meaning of the operations is precisely specified by means of laws, while the internal structure of the ADT, i.e., the representation of its values and the definition of its associated operations are left open; there is no need to define the internal structure of an ADT at the time of defining it.
- An ADT and its associated operations are implemented by a CDT and the operations associated with it, which, however, are kept invisible to a user of the ADT.
- In general, an ADT can be implemented by various CDTs, which can be chosen for simplicity, performance, etc.

...the Haskell means of choice for defining and implementing ADTs are modules hiding their CDT implementations.

Why Abstract Data Types?

...by introducing a level of indirection between specification and implementation of a data type, we achieve:

- Separation of concerns: Separation of specification (interface and behaviour specification) and implementation of a data type (in terms of a CDT and CDT operations matching the ADT operations).
- Information hiding: No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.
- Security: CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated using the ADT operations implemented by their CDT counterparts.

Defining and Implementing an ADT

...is technically a three-stage approach of specification, implementation, and verification:

- Specification (user-visible)
  - Interface Specification: Signatures of ADT operations
  - Behaviour Specification: Laws for ADT operations
- Implementation (user-invisible)
  - Implementing the ADT values in terms of a CDT
  - Implementing the ADT operations as CDT operations
- Verification
  - Specification: Proving that the ADT laws are consistent and complete (proof obligation of the ADT specificator)
  - Implementation: Proving that the implemented CDT operations are sound, i.e., satisfy the ADT laws (proof obligation of the CDT implementor)

Benefits of Abstract Data Type Definitions

...supporting programming-in-the large:

- Enabling modular program development by separating the responsibilities for specifying and implementing a data type and the operations associated with it.

...supporting reusability and maintainability:

- If non-functional requirements for an ADT implementation change or evolve over time, a current CDT implementation of the ADT and its operations can easily be replaced by a new one fitting better to the new requirements as long as the new CDT implementation satisfies the interface and behaviour specification of the ADT.

In the following

…we will demonstrate this considering ADT definitions and implementations for

- Stacks
- Queues
- Priority Queues
- Tables

Chapter 8.2

Stacks

Interface Specification

...of the ADT stack, named `Stack` (user-visible):

```
module Stack (Stack,emptyS,is_emptyS,push,pop,top)
                                                 where

-- Interface Spec.: Signatures of stack operations
emptyS    :: Stack a
is_emptyS :: Stack a -> Bool
push      :: a -> Stack a -> Stack a
pop       :: Stack a -> Stack a
top       :: Stack a -> a

-- Behaviour Spec.: Laws for stack operations
(1) thru (6)                 -- cf. next slide; laws
                             -- must be ensured by
                             -- any implementation.
```

Behaviour Specification

...of the stack operations of the ADT stack (user-visible):

```
Behaviour Spec.: Laws for stack operations

1) is_emptyS emptyS      == True
2) is_emptyS (push v s) == False
3) top emptyS           == undef
4) top (push v s)       == v
5) pop emptyS           == undef
6) pop (push v s)       == s
```

Note: The above laws enforce a last-in/first-out (LIFO) behaviour of stacks.

Implementation A

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

```
data Stack a      = Empty | Stk a (Stack a)

emptyS            = Empty

is_emptyS Empty = True
is_emptyS _       = False

push x s          = Stk x s

pop Empty         = error "Stack is empty"
pop (Stk _ s)     = s

top Empty         = error "Stack is empty"
top (Stk x _)     = x
```

Implementation B

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

```
newtype Stack a     = Stk [a]

emptyS              = Stk []

is_emptyS (Stk []) = True
is_emptyS (Stk _)  = False

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

pop (Stk [])        = error "Stack is empty"
pop (Stk (_:xs))    = Stk xs

top (Stk [])        = error "Stack is empty"
top (Stk (x:_))     = x
```

"Implementation" C

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

```
type Stack a = [a]

emptyS       = []

is_emptyS [] = True
is_emptyS _  = False

push x xs    = (x:xs)

pop []       = error "Stack is empty"
pop (_:xs)   = xs

top []       = error "Stack is empty"
top (x:_)    = x
```

Verification

Specificator and implementor of the ADT stack can prove, respectively:

Lemma 8.2.1 (Consistency and Completeness)

The 6 laws of the behaviour specification of the ADT stack are consistent and complete.

Lemma 8.2.2 (Soundness)

Implementations A and B (and C) satisfy the 6 laws of the behaviour specification of the ADT stack .

Critical Remark

...on "Implementation" C of stacks as an

- alias type of predefined lists: `type Stack a = [a]`

Obvious (but actually only apparent) benefit of implementing stacks as predefined lists:

- Even less conceptual overhead than for stacks implemented as a new type `newtype Stack a = Stk [a]` where the constructor `Stk` needs to be handled by the implementations of the stack operations.

But

Security is broken and lost!

- All predefined operations on lists are available on stacks (not just the 5 ADT operations of stack).

Worse

- Many of the predefined operations on lists (reversal, element picking, etc.) are not even meaningful for stacks.
- Even hiding the implementation in a module can not prevent the application of such meaningless operations to stacks but requires to explicitly abstain from them.

Hence

- "Implementation" C violates the spirit of an ADT implementation and should not be considered a reasonable and valid implementation of the ADT stack.

Chapter 8.3

Queues

Interface Specification

...of the ADT queue, named Queue (user-visible):

```
module Queue (Queue,emptyQ,is_EmptyQ,
              enQ,deQ,frontQ) where

-- Interface Spec.: Signatures of queue operations
emptyQ    :: Queue a
is_emptyQ :: Queue a -> Bool
enQ       :: a -> Queue a -> Queue a
deQ       :: Queue a -> Queue a
frontQ    :: Queue a -> a

-- Behaviour Spec.: Laws for queue operations
(1) thru (6)                 -- cf. next slide; laws
                             -- must be ensured by
                             -- any implementation.
```

Behaviour Specification

...of the queue operations of the ADT queue (user-visible):

Behaviour Spec.: Laws for queue operations:

```
1) is_emptyQ emptyQ     == True
2) is_emptyQ (enQ v q) == False
3) frontQ emptyQ        == undef
4) frontQ (enQ v q)     == if is_emptyQ q
                            then v
                            else frontQ q
5) deQ emptyQ           == undef
6) deQ (enQ v q)        == if is_emptyQ q
                            then emptyQ
                            else enQ ((deQ q) v)
```

Note: The above laws enforce a first-in/first-out (FIFO) behaviour of queues.

Implementation A

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

```
newtype Queue a  = Q [a]

emptyQ           = Q []

is_emptyQ (Q []) = True
is_emptyQ _      = False

enQ x (Q q)      = Q (q ++ [x])

deQ (Q [])       = error "Queue is empty"
deQ (Q (_:xs))   = Q xs

frontQ (Q [])    = error "Queue is empty"
frontQ (Q (x:_)) = x
```

Implementation B

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

```
newtype Queue a          = Q ([a],[a])
```

front rear (in reverse order)
of the queue

```
emptyQ                   = Q ([],[])
is_emptyQ (Q ([],[]))    = True
is_emptyQ _              = False
enQ x (Q ([],[]))        = Q ([x],[])
enQ y (Q (xs,ys))        = Q (xs,y:ys)
deQ (Q ([],[]))          = error "Queue is empty"
deQ (Q ([],ys))          = Q (tail(reverse ys),[])
deQ (Q (x:xs,ys))        = Q (xs,ys)
frontQ (Q ([],[]))       = error "Queue is empty"
frontQ (Q ([],ys))       = last ys
frontQ (Q (x:xs,ys))     = x
```

Verification

Specificator and implementor of the ADT queue can prove, respectively:

Lemma 8.3.1 (Consistency and Completeness)

The 6 laws of the of the behaviour specification of the ADT queue are consistent and complete.

Lemma 8.3.2 (Soundness)

Implementations A and B satisfy the 6 laws of the behaviour specification of the ADT queue.

Homework 8.3.3

Implementation B of the ADT queue is more efficient than implementation A. Why?

Chapter 8.4

Priority Queues

Interface/Behaviour Specification

...of the ADT priority queue, named PQueue (user-visible):

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

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

-- Behaviour Spec.: Laws for priority queue operations
   ...Homework!
```

Note: Each entry of a priority queue has a priority associated with it. The dequeue operation always removes the entry with the highest (or lowest) priority, which is ensured by the enqueue operation, which places a new element according to its priority in a queue.

Implementation

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

```
newtype PQueue a    = PQ [a]

emptyPQ             = PQ []

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

enPQ x (PQ pq)      = PQ (insert x pq)
 where
  insert x []                 = [x]
  insert x r@(e:r') | x <= e = x:r -- the smaller the
                                   -- higher the priority
                    | otherwise = e:insert x r'

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

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

Verification

Specificator and implementor of the ADT priority queue need to show, respectively:

- The laws of the behaviour specification of the ADT priority queues are consistent and complete
- The implementation satisfies the laws of the behaviour specification of the ADT priority queue

...where the specification of the laws was left for homework.

Chapter 8.5

Tables

Chapter 8.5.1

Tables as Functions and Lists

Interface/Behaviour Specification

...of the ADT table, named `Table` (user-visible):

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

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

-- Behaviour Spec.: Laws for table operations
   Intuitively:
    -- new_T assoc_list: create a new table and ini-
    --   tialize it with the data of assoc_list.
    -- find_T tab ind: retrieve information stored in
    --   table tab at index ind.
    -- upd_T (ind,val) tab: update the entry of table
    --   tab stored at index ind with value val.
   Details: Homework!
```

Implementation A

...of the ADT table as a function (user-invisible):

```
newtype Table a b = Tbl (b -> a)

new_T assoc_list =
 foldr upd_T
       (Tbl (_ -> eror "Item not found"))
       assoc_list

find_T (Tbl f) index = f index

upd_T (index,value) (Tbl f) = Tbl g
 where g j | j==index  = value
           | otherwise = f j
```

Implementation B

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

```
newtype Table a b = Tbl [(b,a)]

new_T assoc_list = Tbl assoc_list

find_T (Tbl []) i = error "Item not found"
find_T (Tbl ((j,value):r)) index
 | index==j  = value
 | otherwise = find_T (Tbl r) index

upd_T e (Tbl []) = Tbl [e]
upd_T e'@(index,_) (Tbl (e@(j,_):r))
 | index==j    = Tbl (e':r)
 | otherwise   = Tbl (e:r')
 where Tbl r' = upd_T e' (Tbl r)
```

Verification

Specificator and implementor of the ADT table need to show, respectively:

- The laws of the behaviour specification of the ADT table are consistent and complete
- The implementation satisfies the laws of the behaviour specification of the ADT table

...where the specification of the laws was left for homework.

Chapter 8.5.2

Tables as Arrays

Interface/Behaviour Specification

...of the ADT table, named `Table′` (user-visible):

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

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

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

Note: The signatures of the table operations have been enlarged by the context `(Ix b) =>` in order to be prepared for array manipulations.

Implementation

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

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

new_T' assoc_list = Tbl' (array (low,high) assoc_list)
 where indices    = map fst assoc_list
       low        = minimum indices
       high       = maximum indices

find_T' (Tbl' a) index          = a ! index

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

Note

- `new_T'` takes an association list of index/value pairs and returns the corresponding table.

  To this end, `new_T'` determines first the list of indices `indices` of association list `assoc_list`, and based on this the boundaries of the new table array by computing the minimum `low` and the maximum `high` index of `assoc_list`; afterwards it constructs the new table array applying the function `array` to the pair of array bounds `(low,high)` and association list `assoc_list`.
- `find_T'` and `upd_T'` are used to retrieve and update values in the table array, respectively. Note that `find_T'` returns a system error, not a user error, when applied to an invalid index.

Verification

Specificator and implementor of the ADT table need to show, respectively:

- The laws for table are consistent and complete
- The implementation satisfies the laws of the ADT operations of the ADT table

...whose specification was left for homework here.

Chapter 8.6

Displaying ADT Values in Haskell

Displaying ADT Values

...is often necessary but requires some special care, especially in Haskell.

The reasons for this are twofold:

- ADT values can only be accessed using the ADT operations. Usually, it is crude and cumbersome to display all values of a complex ADT value like a stack or a queue using only the ADT operations, e.g., by completely popping a whole stack.
- Displaying ADT values straightforwardly in terms of their CDT representations can reveal the internal structure of the CDT breaking the ADT principles of information hiding and (possibly) security.

In Haskell

...breaking the principles of information hiding and (possibly) security always happens if the CDT implementing an ADT is made an instance of the type class Show using an automatic

- `deriving`-clause

which is demonstrated next considering stacks for illustration.

Displaying Stacks using deriving-Clauses

...is unsafe:

```
data Stack a    = Empty
                | Stk a (Stack a) deriving Show

newtype Stack a = Stk [a] deriving Show

type Stack a    = [a] -- Lists are instance of Show;
                      -- hence, no deriving clause
                      -- required.
```

because displaying stack values reveals their internal structure:

```
push 3 (push 2 (push 1 emptyS))
  ->> Stk 3 (Stk 2 (Stk 1 Empty))

push 3 (push 2 (push 1 emptyS))
  ->> Stk [3,2,1]

push 3 (push 2 (push 1 emptyS))
  ->> [3,2,1] ->> (3:2:1:[])
```

Note on Information Hiding and Security (1)

Information hiding

- is broken for all three implementation variants as algebraic type, new type, and type alias: Displaying stack values discloses their internal structure and data constructors.

Security

- is broken for the variant as type alias: All list operations are immediately available to create, access, and manipulate stack values using arbitrary list operations. Therefore, type aliases of basic types are not considered valid ADT implementations.
- is preserved for the variants as algebraic type and new type: This is because the data value constructors `Empty` and `Stk` are not exported from the module. A user of the module can thus not use or create a stack value by any other way than the operations exported by the module.

Note on Information Hiding and Security (2)

This holds analogously for other ADT implementations:

Stacks

```
data Stack a    = Empty
                | Stk a (Stack a) deriving Show
newtype Stack a = Stk [a] deriving Show
type Stack a    = [a]
```

Queues and Priority Queues

```
newtype Queue a  = Q [a] deriving Show
newtype PQueue a = PQ [a] deriving Show
```

Tables

```
newtype Table a b = Tbl [(b,a)] deriving Show
newtype Table a b = Tbl (Array b a) deriving Show
```

...straightforward and easy but unsafe and (possibly) insecure.

Displaying Stacks using instance-Decl.'s (1)

...the safe and secure, and thus recommended way for displaying ADT values, here stacks:

A)
```
instance (Show a) => Show (Stack a) where
  showsPrec _ Empty str = showChar ‘-‘ str
  showsPrec _ (Stk x s) str
    = shows x (showChar ‘|‘ (shows s str))
```

B)
```
instance (Show a) => Show (Stack a) where
  showsPrec _ (Stk []) str = showChar ‘-‘ str
  showsPrec _ (Stk (x:xs)) str
    = shows x (showChar ‘|‘ (shows (Stk xs) str))
```

C)
```
instance (Show a) => Show (Stack a) where
  showsPrec _ [] str = showChar ‘-‘ str
  showsPrec _ (x:xs) str
    = shows x (showChar ‘|‘ (shows xs str))
```

Displaying Stacks using `instance`-Decl.'s (2)

This way, the very same output for all 3 implementations:

```
push 3 (push 2 (push 1 emptyS)) ->> 3|2|1|-
```

No implementation details about the internal data structure are disclosed:

- Independently of the chosen implementation A, B, (or C), the output is the same.
- Hence, the actually chosen implementation of the ADT `Stack` remains hidden. It is not disclosed to the user (of the module).

Note: The first argument of `showsPrec` is an unused precedence value.

Displaying Tables Represented as Functions

...note that there is no general meaningful way to display a function. An instance declaration for

```
newtype Table a b = Tbl (b -> a)
```

for the type class `Show` could thus be chosen minimal/trivial:

```
instance Show (Table a b) where
  showsPrec _ _ str = showString "<<A Table>>" str
```

Chapter 8.7

Summary

Abstract Data Types

...are not a first-class citizen in Haskell.

Nonetheless, specifying and implementing ADTs using modules ensures all three design goals strived for with ADTs:

- Separation of concerns: Separation of specification (interface and behaviour specification) and implementation of a data type (in terms of a CDT and CDT operations matching the ADT operations).
- Information hiding: No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.
- Security: CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated by using the ADT operations implemented by their CDT counterparts.

Note

Due to the limitation of the module concept in Haskell, the

- behaviour specification of an ADT can only be provided in terms of comments.

If ADT values need to be displayed, this can be done by

- by making the underlying CDT a member of the type class `Show`.

This should always and only be done by means of an explicit

- `instance`-declaration

since a (more convenient) `deriving`-clause would reveal the internal representation of the CDT values, especially the data constructors of the CDT breaking the information hiding principle of ADTs (though the constructors could not be used by a user since they are not exported from the module).

Benefits of Using Abstract Data Types

...evolve directly from the 'by-design built-in' ADT properties:

- Separation of concerns, i.e. the separation of the specification and implementation of a data type

enables

- Information hiding: Only the interface and the behaviour specification of the ADT are publicly known; its implementation as a CDT and operations on it are hidden.

This ensures:

- Security of the data (structure) and its data values from uncontrolled, unintended, or not permitted access.

Altogether, this enables:

- Simple exchangeability of the CDT implementation of an ADT (e.g., simplicity vs. scalability/performance).
- Modularization and programming-load sharing supporting programming-in-the-large.

Relevance of Abstract Data Types

...there are many more examples of data structures, which can be specified and implemented in terms of abstract data types in order to benefit from the built-in ADT properties such as separation of concerns, information hiding, security, exchangeability, modularity, etc., including

- Sets
- Heaps
- Trees (binary search trees, balanced trees,...)
- ...

and also

- Arrays

as illustrated next.

Arrays as Abstract Data Type in Haskell (1)

```
module Array (
        module Ix, -- export all of Ix (for convenience)
        Array, array, listarray (!), bounds, indices,
        elems, assocs, accumArray, (//),
        accum, ixmap ) where

import Ix
infixl 9 !, // ... -- Operator precedence
data (Ix a) => Array a b = ... -- Abstract

array     :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!)       :: (Ix a) => Array a b -> a -> b
bounds    :: (Ix a) => Array a b (a,a)
indices   :: (Ix a) => Array a b -> [a]
elems     :: (Ix a) => Array a b -> [b]
assocs    :: (Ix a) => Array a b -> [(a,b)]
```

Arrays as Abstract Data Type in Haskell (2)

```
accumArray :: (Ix a) => (b -> c -> b) -> b
                          -> (a,a) -> [(a,c)] -> Array a b
(//)       :: (Ix a) => Array a b -> [(a,b)]
                                     -> Array a b
accum      :: (Ix a) => (b -> c -> b) -> Array a b
                                 -> [(a,c)] -> Array a b
ixmap      :: (Ix a, Ix b) => (a,a) -> (a -> b)
                               -> Array b c -> Array a c

instance Functor (Array a) where...
instance (Ix a, Eq b)  => Eq (Array a b) where...
instance (Ix a, Ord b) => Ord (Array a b) where...
instance (Ix a, Show a, Show b)
                => Show (Array a b) where...
instance (Ix a, Read a, Read b)
                => Read (Array a b) where...
```

Arrays as Abstract Data Type in Haskell (3)

For the definition of the functions and instance declarations of the module `Array`, see:

- Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 173-178, 2003. (Chapter 16, Arrays)

Chapter 8.8

References, Further Reading

Chapter 8: Further Reading (1)

- Manoochehr Azmoodeh. *Abstract Data Types and Algorithms*. Macmillan Education, 1988.
- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2nd edition, 1998. (Chapter 8, Abstract data types)
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 8.4, Abstract types)
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Chapter 10, Arrays, Listen und Stacks)
- F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.

Chapter 8: Further Reading (2)

- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.5, Abstract Types and Modules)

- Gerhard Goos, Wolf Zimmermann. *Programmiersprachen*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 515-562, 2006. (Kapitel 2.1, Methodische Grundlagen: Abstrakte Datentypen, Grundlegende abstrakte Datentypen)

- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.

- John V. Guttag, James J. Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.

Chapter 8: Further Reading (3)

- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.
- Rachel Harrison. *Abstract Data Types in Standard ML*. J. Wiley, 1993.
- Chris Okasaki. *Simple and Efficient Purely Functional Queues and Dequeues*. Journal of Functional Programming 5(4):583-592, 1995.
- Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Chapter 8: Further Reading (4)

- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 14.1, Abstrakte Datentypen; Kapitel 14.3, Generische abstrakte Datentypen; Kapitel 14.4, Abstrakte Datentypen in ML und Gofer; Kapitel 15.3, Ein abstrakter Datentyp für Sequenzen)
- Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. (Chapter 16, Arrays)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract Data Types)

Chapter 8: Further Reading (5)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 16, Abstract data types)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 16, Abstract data types)

Chapter 9

Monoids

Chapter 9.1

Motivation

Types

...equipped with an associative operation and a left-unit and a right-unit like

- lists with concatenation `(++)` and unit `[]`

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)(associative)
          [] ++ xs = xs              (left-unit)
          xs ++ [] = xs             (right-unit)
```

- `Bool` with conjunction `(&&)` and unit `True`

```
(b1 && b2) && b3 = b1 && (b2 && b3)(associative)
         True && b = b               (left-unit)
         b && True = b              (right-unit)
```

should be made instances of the type class `Monoid`.

Chapter 9.2

The Type Class Monoid

The Type Class Monoid

Type Class Monoid

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

...monoids are instances of the type class Monoid (and hence types), which obey the monoid laws:

Monoid Laws

```
mempty `mappend` x             = x                  (MoL1)
x `mappend` mempty             = x                  (MoL2)
(x `mappend` y) `mappend` z  =
  x `mappend` (y `mappend` z)                       (MoL3)
```

Intuitively

Monoids are types which provide

- a binary operation `mappend`, a value `mempty`, and a function `mconcat`.

The monoid laws

- MoL1 and MoL2 require that `mempty` is a left-unit and a right-unit of `mappend`.
- MoL3 requires that `mappend` is associative.
- The function `mconcat` takes a list of monoid values and reduces them to a single monoid value by using `mappend`.

Note: It is a programmer obligation to prove that their instances of `Monoid` satisfy the monoid laws.

Note

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

Chapter 9.3

Monoid Examples

Chapter 9.3.1

Lists as Monoid

Lists as Monoid

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

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

Lemma 9.3.1.1 (Monoid Laws for [a])

For every instance of type a, the instance [a] of Monoid satisfies the three monoid laws MoL1, MoL2, and MoL3, and is hence a monoid, the so-called list monoid.

Examples

...evaluating some terms for illustration:

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

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

 mempty ->> []

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

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

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

Note

...commutativity of mappend is not required by the monoid laws. E.g.:

```
"Semester " `mappend` "Holiday"
      ->> "Semester Holiday"
```

is different from

```
"Holiday " `mappend` "Semester"
      ->> "Holiday Semester"
```

Chapter 9.3.2

Numerical Types as Monoids

Numerical Types (and Boolean) as Monoids

Numerical types (as well as the Boolean type) are equipped with more than one operation that behave as required for the monoid operation `mappend`. E.g.:

- `*` and `+` for numerical types
- `||` and `&&` for `Bool`

Hence, we will make use of newtype declarations for types of

- numerical and Boolean values

to allow more than one monoid instance for them.

Moreover, we will use

- record syntax

to get selector functions for free (cf. Chapter 5.4, LVA 185.A03 Funktionale Programmierung).

The Sum and Product Monoids (1)

...the sum monoid of numerical types:

```
newtype Sum a = Sum { getSum :: a }
 deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
 mempty  = Sum 0
 Sum x `mappend` Sum y = Sum (x+y)
```

...the product monoid of numerical types:

```
newtype Product a = Product { getProduct :: a }
 deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Product a) where
 mempty  = Product 1
 Product x `mappend` Product y = Product (x*y)
```

The Sum and Product Monoids (2)

Lemma 9.3.2.1 (Monoid Laws for `Sum` and `Product`)

For every numerical instance of type `a`, the instances `(Sum a)` and `(Product a)` of `Monoid` satisfy the three monoid laws MoL1, MoL2, and MoL3, and are hence monoids, the so-called product and sum monoids.

Examples

...evaluating some terms for illustration:

```
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

getProduct . mconcat . map Product $ [3,7,11] ->> 231
getSum . mconcat . map Sum $ [3,7,11] ->> 21

Product 3 `mappend` mempty ->> Product 3
getSum $ mempty `mappend` Sum 3 ->> 3
```

Chapter 9.3.3

Bool as Monoid

The All and Any Monoids (1)

...the all monoid of Bool:

```
newtype All = All { getAll :: Bool }
 deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid All where
 mempty  = All True
 All x `mappend` All y = All (x && y)
  -- 'All' because True if every argument is true.
```

...the any monoid of Bool:

```
newtype Any = Any { getAny :: Bool }
 deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid Any where
 mempty  = Any False
 Any x `mappend` Any y = Any (x || y)
  -- 'Any' because True if some argument is true.
```

The All and Any Monoids (2)

Lemma 9.3.3.1 (Monoid Laws for `All` and `Any`)

The instances `All` and `Any` of class `Monoid` satisfy the three monoid laws MoL1, MoL2, and MoL3, and are hence monoids, the so-called all and any monoids.

Examples

...evaluating some terms for illustration:

```
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
```

Remarks on Numerical and Boolean Monoids

Note:

- For the monoids `(Product a)`, `(Sum a)`, `Any`, and `All` the monoid operation `mappend` is both associative and commutative.
- For most instances of the type class `Monoid`, however, this does not hold (and need not to hold). Two such examples are the list monoid `[a]` and the ordering monoid `Ordering` considered next.

Chapter 9.3.4

Ordering as Monoid

Ordering as Monoid (1)

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

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

Note:

- The definition of the operation `mappend` induces an 'alphabetical' comparison of two list arguments.
- The operation `mappend` fails to be commutative for the ordering monoid `Ordering`:

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

Ordering as Monoid (2)

Lemma 9.3.4.1 (Monoid Laws for `Ordering`)

The instance `Ordering` of class `Monoid` satisfies the three monoid laws MoL1, MoL2, and MoL3, and is hence a monoid, the so-called ordering monoid.

Examples (1)

...showing some useful applications of mappend.

Note, the two definitions of lengthCompare w/ and w/out mappend:

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

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

...are equivalent as can be verified by means of the properties of the monoid operation mappend.

Examples (2)

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

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

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

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

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

Examples (3)

...further comparison criteria can easily be added and prioritized.

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

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

As expected we get:

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

Chapter 9.4

Summary and Looking ahead

Summary

Monoids are most useful for defining

- folds over values of various data structures

since folding requires an associative operation.

While for

- lists

folding seems obvious, it is possible for the values of many other data structures, too, e.g.

- trees

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

Looking ahead: Type Constructor Classes (1)

The Type Constructor Class Foldable:

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

Note:

- f and t are applied to type variables, here a and b. Hence, f and t are (1-ary) type constructors, not types.
- Foldable is thus a type constructor class, not just a type class.
- The operations foldl and foldr of Foldable generalize folding of lists to folding of values of other 'foldable' data structures.

Looking ahead: Type Constructor Classes (2)

...the type constructor [] for lists is one important instance of Foldable:

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

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

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

Chapter 9.5

References, Further Reading

Chapter 9: Further Reading

- Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 13.4.3, Defining New Type Classes for Behaviors)
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 12, Monoids)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 13, Data Structures – Monoids)

Chapter 10

Functors

Types

...whose values can be mapped over compositionally, with a neutral element, like

- lists with mapL and id

```
g :: a -> b, h :: b -> c

mapL g []       = []
mapL g (x:xs)   = (g x) : mapL g xs

mapL (h . g) xs = mapL h (mapL g xs)   (compositional)
mapL id xs      = xs                 (neutral element)
```

- trees with mapT and id

```
g :: a -> b, h :: b -> c

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

mapT g (Leaf v)     = Leaf (g v)
mapT g (Node v l r) = Node (g v) (mapT g l) (mapT g r)

mapT (h . g) t = mapT h (mapT g t)     (compositional)
mapT id t      = t                   (neutral element)
```

should be made an instance of type constructor class Functor.

Chapter 10.1

Motivation

Mapping

...over values is a typical and recurring task. Recall:

- Lists

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

- Trees

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

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

Higher-Order Type (Constructor) Classes

..the similarity of tasks performed by functions like

- `mapL`
- `mapT`

suggests bundling all types whose values can be mapped over in a unique type class offering an (over-loaded) function

- mapGeneric

which covers `mapL`, `mapT`, and many more:

- Type class `Functor`

Note that `Functor` is a representative of a new kind of type classes, higher-order type classes or

- type (constructor) classes.

Chapter 10.2

The Type Constructor Class Functor

The Type Constructor Class Functor

Type Constructor Class Functor

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

...functors are instances of the type constructor class Functor (and hence 1-ary type constructors), which obey the functor laws:

Functor Laws

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

Note: It is a programmer obligation to prove that their instances of Functor satisfy the functor laws.

Note

...argument `f` of `Functor` is applied to type variables. Hence:

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

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

The functor laws ensure:

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

Class Functor, Functor Laws in more Detail

...with added type information:

Class Functor

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

Functor Laws

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

(id over-loaded!)

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

Curried and Uncurried View of fmap

Curried view: `fmap` takes

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

Example:

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

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

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

Uncurried view: `fmap` takes

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

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

Type Classes vs. Type Constructor Classes (1)

Recall the definition of the type class `Monoid` to compare it with the type constructor class `Functor`:

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

Note:

- The argument `m` of `Monoid` is a type variable. Functions declared in `Monoid` operate on values of type `m`; `m` itself does not operate on anything.
- This holds for every type class; recall the definitions of type classes we considered so far: `Eq`, `Ord`, `Num`, `Enum`,...

Type Classes vs. Type Constructor Classes (2)

Type classes and type constructor classes are conceptually equal. They differ in the type of their members:

- Type constructor classes (`Foldable`, `Functor`, `Monad`, `Arrow`,...) have
  - type constructors (e.g., `Tree`, `[]`, `(,)`, `(->)`,...) as members.
- Type classes (`Eq`, `Ord`, `Num`, `Monoid`,...) have
  - types (e.g., `Tree a`, `[] a`, `(,) a a`, `(->) a a`,...) as members.

Type constructors are

- maps, which construct new types from given types.

  Examples: Tuple constructors `(,)`, `(,,)`, `(,,,)`; list constructor `[]`; map constructor `(->)`; input/output constructor `IO`,...

The List and Tree Functors [] and Tree (1)

...making the 1-ary type constructors [] and Tree for lists and trees, respectively, instances of the type constructor class Functor:

```
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 (over-loaded), as a
  - type constructor in: `instance Functor [] where...`
  - value of some list type in: `fmap g [] = []`.
- The declarations `instance Functor [a] where...`, `instance Functor (Tree a) where...` would not be correct, since `[a]` and `(Tree a)` denote types, no type constructors.

The List and Tree Functors [] and Tree (2)

Lemma 10.2.1 (Functor Laws for [] and Tree)

The instances [] and Tree of the type constructor class Functor satisfy the two functor laws FL1 and FL2, respectively, and hence, are functors, the so-called list and tree functor.

The List and Tree Functors [] and Tree (3)

The instance declarations for [] and Tree could have been equivalently but more concisely given as follows:

```
instance Functor [] where
  fmap = mapL                  -- user-defined mapL

instance Functor [] where
  fmap = map                   -- predefined map

instance Functor Tree where
  fmap = mapT                  -- user-defined mapT
```

The List and Tree Functors [] and Tree (4)

Examples:

```
ms = [1..5]

fmap (*2) ms ->> [2,4,6,8,10]
fmap (^3) ms ->> [1,8,27,64,125]
fmap (3^) ms ->> [3,9,27,81,243]

t = Node 2 (Node 3 (Leaf 5) (Leaf 7)) (Leaf 11)

fmap (*2) t
 ->> Node 4 (Node 6 (Leaf 10) (Leaf 14)) (Leaf 22)

fmap (^3) t
 ->> Node 8 (Node 27 (Leaf 125) (Leaf 343))
            (Leaf 1331)
fmap (3^) t
 ->> Node 9 (Node 27 (Leaf 243) (Leaf 2187))
            (Leaf 177147)
```

Note

...the operation `fmap` of the type constructor class `Functor` is

- the (over-loaded) generic map `mapGeneric`

that we were looking and striving for.

Members of the type constructor class `Functor` can be

- pre-defined
- user-defined

1-ary type constructors.

Examples of Predefined Type Constructors

...of different arity:

- 1-ary type constructors: `[]`, `Maybe`, `IO`, ...
- 2-ary type constructors: `( , )`, `(->)`, `Either`, ...
- 3-ary type constructors: `( , , )`, ...
- 4-ary type constructors: `( , , , )`, ...
- ...

Note:

- Only 1-ary type constructors are instance candidates of `Functor`. This can be also partially evaluated type constructors of higher arity, e.g., `(Either a)`, `((->) r)`.
- Considering types as 0-ary type constructors shows the conceptual coincidence of type classes and type constructor classes.

Notational Remark

Recall, the following notations are equivalent:

- `(a,b)` is equivalent to `(,) a b`
  `(a,b,c)` is equivalent to `(, ,) a b c`, etc.
- `[a]` is equivalent to `[] a`
- `a -> b` is equivalent to `(->) a b`
- `T a b` is equivalent to `((T a) b)` (i.e., associativity to the left as for function application)

Example

...the signatures of

```
fac :: Int -> Int
list2pair :: [a] -> (a,a)
```

can equivalently be written in the form:

```
fac :: (->) Int Int
list2pair :: [] a -> (a,a)
list2pair :: [a] -> (,) a a
list2pair :: (->) [a] (a,a)
list2pair :: [] a -> (,) a a
...
list2pair :: (->) ([] a) ((,) a a)
```

However, we are more familar with the 'classical' forms, which may thus appear more easily comprehensible.

Chapter 10.3

Predefined Functors

Chapter 10.3.1

The Identity Functor

The Identity Functor

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

```
newtype Id a = Id a

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

Lemma 10.3.1.1 (Functor Laws for Id)

The instance Id of the type constructor class Functor satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called identity functor.

Chapter 10.3.2

The Maybe Functor

The Maybe Functor

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

```
data Maybe a = Nothing | Just a

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

Lemma 10.3.2.1 (Functor Laws for Maybe)

The instance Maybe of the type constructor class Functor satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called maybe functor.

Examples

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

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

Anti-Example: Invalid Functor Instance (1)

...consider the type Maybe_with_counter, which is almost like Maybe but whose Just values contain an additional Int value which shall be used for counting the number of applications of fmap:

```
data Maybe_with_counter a
  = Nothing_wc | Just_wc Int a deriving Show
```

...making Maybe_with_counter an instance of Functor:

```
instance Functor Maybe_with_counter where
  fmap g Nothing_wc = Nothing_wc
  fmap g (Just_wc counter x) = Just_wc (counter+1) (g x)
```

We will show: The Maybe_with_counter instance of Functor

- violates functor law FL1.

Hence, Maybe_with_counter is an invalid instance of Functor and thus an anti-example.

Anti-Example: Invalid Functor Instance (2)

```
Nothing_wc              :: Maybe_with_counter a
Just_wc 0 "fun"         :: Maybe_with_counter [Char]
Just_wc 100 [1,2,3]     :: Maybe_with_counter [Int]

Nothing_wc              ->> Nothing_wc
Just_wc 0 "fun"         ->> Just_wc 0 "fun"
Just_wc 100 [1,2,3]     ->> Just_wc 100 [1,2,3]

fmap (++ "prog") Nothing_wc
  ->> Nothing_wc

fmap (++ "prog") (Just_wc 0 "fun")
  ->> Just_wc 1 "funprog"

fmap (++ "prog") (fmap (++ " ") (Just_wc 0 "fun"))
  ->> Just_wc 2 "fun prog"
```

...while everything is absolutely fine with these examples...

Anti-Example: Invalid Functor Instance (3)

...evaluating the expressions

```
fmap id (Just_wc 0 "fun")
```

and

```
id (Just_wc 0 "fun")
```

yield different values:

```
fmap id (Just_wc 0 "fun") ->> Just_wc 1 "fun"
id (Just_wc 0 "fun")      ->> Just_wc 0 "fun"
```

Hence, functor law FL1 is violated: Equality `fmap id = id` does not hold for the `Maybe_with_counter` instance. Thus:

Corollary 10.3.2.2 (Invalid Instance)

`Maybe_with_counter` is not a valid instance of `Functor`.

Chapter 10.3.3

The List Functor

The List Functor

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

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

Lemma 10.3.3.1 (Functor Laws for [])

The instance [] of the type constructor class Functor satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called list functor.

Chapter 10.3.4

The Input/Output Functor

The Input/Output Functor

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

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

Lemma 10.3.4.1 (Functor Laws for IO)

The instance IO of the type constructor class Functor satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called input/output functor.

Examples (1)

...the two versions of program main

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

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

which differ in using and not using fmap are equivalent.

Examples (2)

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

The expressions

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

and

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

have the same input/output effect.

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

Chapter 10.3.5

The Either Functor

The Either Functor

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

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

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

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

Lemma 10.3.5.1 (Functor Laws for (Either a))

The instance (Either a) of the type constructor class Functor satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called either functor.

Examples

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

Homework

Consider the following instance declaration for (Either a):

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

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

Would this instance declaration be meaningful?

Think about the constraints the above instance declaration imposes on the types which are eligible for a and b.

Chapter 10.3.6

The Map Functor

The Map Functor

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

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

Note: `Either` and `(->)` are both 2-ary type constructors, i.e., have two arguments. Hence, only the partially evaluated type constructors `(Either a)` and `((->) d)` can be made instances of `Functor`, since they are 1-ary type constructors.

Lemma 10.3.6.1 (Functor Laws for `((->) d)`)

The instance `((->) d)` of the type constructor class `Functor` satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called map functor.

The Map Functor in more Detail

...with added type information:

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

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

Note: `fmap` defined (as above) by

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

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

The Instance Declaration of the Map Functor

...reconsidered.

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

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

Notes on the Map Functor

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

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

specializes to:

```
fmap :: (a -> b) -> (((->) d) a) -> (((->) d) b)
```

Using infix notation for `(->)`, this can equivalently be written as:

```
fmap :: (a -> b) -> (d -> a) -> (d -> b)
```

where `fmap` can be implemented by:

$$\texttt{fmap}\ \underbrace{\texttt{g}}_{::\ \texttt{a -> b}}\ \underbrace{\texttt{h}}_{::\ \texttt{d -> a}}\ =\ \underbrace{(\underbrace{\texttt{g}}_{::\ \texttt{a -> b}}\ \underbrace{\texttt{.}}_{::\ \texttt{(a -> b) -> (d -> a) -> (d -> b)}}\ \underbrace{\texttt{h}}_{::\ \texttt{d -> a}})}_{::\ \texttt{d -> b}}$$

Examples (1)

```
Main>:t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a

fmap (*3) (+100) 1          ->>  303

(*3) `fmap` (+100) $ 1      ->>  303
(*3)    .   (+100) $ 1      ->>  303

fmap (show . (*3)) (+100) 1 ->> "303"
```

Note: Using `fmap` as an infix operator emphasizes the equality of `fmap` and functional composition `(.)` for the map functor `((->) d)`.

Examples (2)

...recalling the generic type of fmap:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

we get:

```
Main>:t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a

Main>:t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

where

```
replicate :: Int -> a -> [a]
replicate n x
 | n <= 0      = []
 | otherwise = x : replicate (n-1) x
```

Examples (3)

```
fmap (replicate 3) [1,2,3,4]
 ->> [[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

fmap (replicate 3) (Just 4)
 ->> Just [4,4,4]

fmap (replicate 3) (Right "fun")
 ->> Right ["fun","fun","fun"]

fmap (replicate 3) Nothing
 ->> Nothing

fmap (replicate 3) (Left "fun")
 ->> Left "fun"
```

Examples (4)

Applying fmap to n-ary maps (e.g., (*), (++), \x y z -> ..., ...) instead of 1-ary maps (e.g., replicate 3, (*3), (+100), ...) as so far, we get:

```
fmap (*) (Just 3) ->> Just ((*) 3)

fmap (++) (Just "fun") :: Maybe ([Char] -> [Char])
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
fmap compare "A list of chars" :: [Char -> Ordering]
fmap (\x y z -> x + y / z) [3,4,5,6]
                :: (Fractional a) => [a -> a -> a]

a = fmap (*) [1,2,3,4] :: [Int -> Int]

fmap (\f -> f 9) a ->> [9,18,27,36]
```

Note

...some of the previous examples showed

- lifting

of a map of type

- `(a -> b)`

to type

- `(f a -> f b)`

by `fmap`. This again shows that `fmap`

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

can be thought of in two ways. As a map which takes a map `g :: a -> b` and

1. lifts `g` to a new function `h :: f a -> f b` operating on functor values ⇝ curried view.
2. a functor value `v :: f a` and maps `g` over `v` ⇝ uncurried view.

Homework

Following the example of the map functor, provide (most general) type information for the following instance declarations of `Functor`:

1. Identity
2. Maybe
3. List
4. Input/Output
5. Either
6. Tree (cf. Chapter 10.2)

Chapter 10.4

The Type Constructor Class Applicative

The Type Constructor Class Applicative

Type Constructor Class Applicative

```
class (Functor f) => Applicative f where
 pure  :: a -> f a                    -- Value ‘lifting’:
                                      -- Making an appli-
                                      -- cative value
 (<*>) :: f (a -> b) -> f a -> f b  -- Mapping over
```

Intuitively

- pure takes a value of any type and returns an applicative value.
- (<*>) takes a functor value, which has a function in it, and another functor value, which has a value in it. It extracts the function from the first functor and maps it over the value of the second one.

The Applicative Laws

...applicatives are instances of the type constructor class `Applicative` (and hence 1-ary functors), which obey the applicative laws:

Applicative Laws

```
pure id <*> v                 = v                       (AL1)
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)       (AL2)
pure g <*> pure x             = pure (g x)              (AL3)
u <*> pure y                  = pure ($ y) <*> u        (AL4)
```

Note: It is a programmer obligation to prove that their instances of `Applicative` satisfy the applicative laws.

Class Applicative and Appl. Laws in Detail

...with added type information:

Class Applicative

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative Laws

$$\underbrace{\underbrace{\texttt{pure}\ \underbrace{\texttt{id}}_{::\ \texttt{a -> a}}}_{::\ \texttt{f (a -> a)}}\ \texttt{<*>}\ \underbrace{\texttt{v}}_{::\ \texttt{f a}}}_{::\ \texttt{f a}} \;=\; \underbrace{\texttt{v}}_{::\ \texttt{f a}} \qquad \text{(AL1)}$$

$$\underbrace{\underbrace{\texttt{pure}\ \underbrace{\texttt{g}}_{::\ \texttt{a -> b}}}_{::\ \texttt{f (a -> b)}}\ \texttt{<*>}\ \underbrace{\texttt{pure}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}}_{::\ \texttt{f a}}}_{::\ \texttt{f b}} \;=\; \underbrace{\texttt{pure}\ (\underbrace{\underbrace{\texttt{g}}_{::\ \texttt{a -> b}}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}}_{::\ \texttt{b}})}_{::\ \texttt{f b}} \qquad \text{(AL3)}$$

An Infix Operator <$> as Alias for fmap

...for a more compelling usage in operation sequences involving both `fmap` and `(<*>)`.

The infix alias `(<$>)` of `fmap` of `Functor`:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
g <$> x = fmap g x
```

Example: Using `(<$>)` as infix operator, we can write:

```
(++) <$> Just "Functional " <*> Just "Programming"
  ->> Just "Functional Programming"
```

instead of the less compelling variants using the prefix operator `fmap`:

```
(fmap (++) Just "Functional ") <*> Just "Programming"
  ->> Just "Functional Programming"
```

...or its infix variant `` `fmap` ``:

```
((++) `fmap` Just "Functional ") <*> Just "Programming"
  ->> Just "Functional Programming"
```

Note

...that defining `(<$>)` by

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

would be valid, too, since the context allows to decide if `f` is used as type constructor (`f`) or as an argument (`f`).

Utility Maps for Applicatives

```
liftA2 :: (Applicative f) =>
              (a -> b -> c) -> f a -> f b -> f c
liftA2 g a b = g <$> a <*> b

sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs

sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

Examples:

```
fmap (\x -> [x]) (Just 4)        ->> Just [4]
liftA2 (:) (Just 3) (Just [4]) ->> Just [3,4]
(:) <$> Just 3 <*> Just 4       ->> Just [3,4]
```

Homework

Provide (most general) type information for the applicative laws AL2 and AL4:

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)     (AL2)
u <*> pure y               = pure ($ y) <*> u    (AL4)
```

Chapter 10.5

Predefined Applicatives

Chapter 10.5.1

The Identity Applicative

The Identity Applicative

...making the 1-ary type constructor Id an instance of the type constructor class Applicative (conceptually the simplest applicative):

```
newtype Id a = Id a

instance Applicative Id where
 pure              = Id
 Id g <*> (Id x) = Id (g x)
```

Note: g plays the role of the applicative functor.

Lemma 10.5.1.1 (Applicative Laws for Id)

The instance Id of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called identity applicative.

The Identity Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

```
instance Applicative Id where
     pure       =       Id
:: a -> Id a       :: a -> Id a

      Id g       <*>       Id x      =      Id (g        x)
   :: (a -> b)             :: a             :: a -> b  :: a
:: Id (a -> b)           :: Id a                 :: b
             :: Id b                           :: Id b
```

Chapter 10.5.2

The Maybe Applicative

The Maybe Applicative

...making the 1-ary type constructor `Maybe` an instance of the type constructor class `Applicative`:

```
instance Applicative Maybe where
 pure                     = Just
 Nothing  <*> _           = Nothing
 (Just g) <*> something = fmap g something
```

Note: `g` plays the role of the applicative functor.

Lemma 10.5.2.1 (Applicative Laws for `Maybe`)

The instance `Maybe` of the type constructor class `Applicative` satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called maybe applicative.

The Maybe Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
fmap  :: (Functor f)     => (a -> b) -> f a -> f b
```

```
instance Applicative Maybe where
```

$$\underbrace{\texttt{pure}}_{::\ \texttt{a -> Maybe a}} \;=\; \underbrace{\texttt{Just}}_{::\ \texttt{a -> Maybe a}}$$

$$\underbrace{\underbrace{\texttt{Nothing}}_{::\ \texttt{Maybe (a -> b)}} \;\texttt{<*>}\; \underbrace{\texttt{\_}}_{::\ \texttt{Maybe a}}}_{::\ \texttt{Maybe b}} \;=\; \underbrace{\texttt{Nothing}}_{::\ \texttt{Maybe b}}$$

$$\underbrace{\underbrace{\texttt{(Just g)}}_{::\ \texttt{Maybe (a -> b)}} \;\texttt{<*>}\; \underbrace{\texttt{something}}_{::\ \texttt{Maybe a}}}_{::\ \texttt{Maybe b}} \;=\; \underbrace{\texttt{fmap}\ \underbrace{\texttt{g}}_{::\ \texttt{a -> b}} \; \underbrace{\texttt{something}}_{::\ \texttt{Maybe a}}}_{::\ \texttt{Maybe b}}$$

Examples (1)

```
Just (+3) <*> Just 9
 ->> fmap (+3) (Just 9)
 ->> Just 12

Just (+3) <*> Nothing
 ->> fmap (+3) Nothing
 ->> Nothing

Just (++ "good ") <*> Just "morning"
 ->> fmap (++ "good ") "morning"
 ->> Just "good morning"

Just (++ "good ") <*> Nothing
 ->> fmap (++ "good ") Nothing
 ->> Nothing

Nothing <*> Just "good "
 ->> Nothing
```

Examples (2)

```
pure (+) <*> Just 3  <*> Just 5
 ->> Just (+) <*> Just 3  <*> Just 5
 ->> (fmap (+) Just 3) <*> Just 5
 ->> Just (3+) <*> Just 5
 ->> Just 8

pure (+) <*> Just 3  <*> Nothing
 ->> Just (+) <*> Just 3  <*> Nothing
 ->> fmap (+) Just 3 <*> Nothing
 ->> Just (3+) <*> Nothing
 ->> fmap (3+) Nothing
 ->> Nothing
```

Examples (3)

```
pure (+) <*> Nothing <*> Just 5
 ->> Just (+) <*> Nothing <*> Just 5
 ->> (fmap (+) Nothing) <*> Just 5
 ->> Nothing <*> Just 5
 ->> Nothing
```

Note: The operator `(<*>)` is left-associative, i.e.:

```
pure (+) <*> Just 3 <*> Just 5 =
                (pure (+) <*> Just 3) <*> Just 5
```

Chapter 10.5.3

The List Applicative

The List Applicative

...making the 1-ary type constructor [] an instance of the type constructor class Applicative:

```
instance Applicative [] where
  pure x     = [x]
  gs <*> xs = [g x | g <- gs, x <- xs]
```

Lemma 10.5.3.1 (Applicative Laws for [])

The instance [] of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called list applicative.

The List Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b

instance Applicative [] where
      pure          x    = [ x ]
 :: a -> [] a    :: a     :: a
                        :: [] a

        gs        <*>   xs     = [  g        x | g <- gs, x <- xs]
 :: [] (a -> b)     :: [] a     :: a -> b :: a
                                     :: b
                                          :: [] b
```

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
Part IV
Chap. 7
Chap. 8
Chap. 9
Chap. 10
10.1
10.2
10.3
10.4
10.5
10.5.1

Examples (1)

```
pure "Hallo" :: String       ->> ["Hallo"]
pure "Hallo" :: Maybe String ->> Just "Hallo"

[(*0),(+100),(^2)] <*> [1,2,3]
 ->> [ f x | f <- [(*0),(+100),(^2)], x <- [1,2,3] ]
 ->> [0,0,0,101,102,103,1,4,9]

[(+),(*)] <*> [1,2] <*> [3,4]
 ->> [ f x | f <- [(+),(*)], x <- [1,2] ] <*> [3,4]
 ->> [(1+),(2+),(1*),(2*)] <*> [3,4]
 ->> [ f x | f <- [(1+),(2+),(1*),(2*)], x <- [3,4] ]
 ->> [4,5,5,6,3,4,6,8]
```

Examples (2)

```
(++) <$> ["yes","no","ok"] <*> ["?",".","!"]
 ->> (fmap (++) ["yes","no","ok"]) <*> ["?",".","!"]
 ->> [("yes"++),("no"++),("ok"++)] <*> ["?",".","!"]
 ->> [f x | f <- [("yes"++),("no"++),("ok"++)],
                x <- ["?",".","!"] ]
 ->> ["yes?","yes.","yes!","no?","no.","no!",
      "ok?","ok.","ok!"]
```

Examples (3)

```
filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
 ->> filter (>50) $ (fmap (*) [2,5,10]) <*> [8,10,11]
 ->> filter (>50) $ [(2*),(5*),(10*)] <*> [8,10,11]
 ->> filter (>50) $ [f x | f <- [(2*),(5*),(10*)],
                           x <- [8,10,11] ]
 ->> filter (>50) $ [16,20,22,40,50,55,80,100,110]
 ->> filter (>50) [16,20,22,40,50,55,80,100,110]
 ->> [55,80,100,110]
```

Examples (4)

...the previous example shows that expressions using list comprehension

```
[x*y | x <- [2,5,10], y <- [8,10,11]]
 ->> [16,20,22,40,50,55,80,100,110]
```

...can alternatively be written using (<$>) and <*>:

```
(*) <$> [2,5,10] <*> [8,10,11]
 ->> [16,20,22,40,50,55,80,100,110]
```

Chapter 10.5.4

The Input/Output Applicative

The Input/Output Applicative

...making the 1-ary type constructor IO an instance of the type constructor class Applicative:

```
instance Applicative IO where
 pure     = return
 a <*> b = do g <- a
              x <- b
              return (g x)
```

Lemma 10.5.4.1 (Applicative Laws for IO)

The instance IO of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called input/output applicative.

The Input/Output Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

```
instance Applicative IO where
```

$$\underbrace{\texttt{pure}}_{::\ \texttt{a -> IO a}} \ = \ \underbrace{\texttt{return}}_{::\ \texttt{a -> IO a}}$$

$$\underbrace{\texttt{a}}_{::\ \texttt{IO (a -> b)}} \ \texttt{<*>} \ \underbrace{\texttt{b}}_{::\ \texttt{IO a}} \ = \ \texttt{do} \ \underbrace{\texttt{g}}_{::\ \texttt{a -> b}} \ \texttt{<-} \ \underbrace{\texttt{a}}_{::\ \texttt{IO (a -> b)}}$$

$$\underbrace{\texttt{x}}_{::\ \texttt{a}} \ \texttt{<-} \ \underbrace{\texttt{b}}_{::\ \texttt{IO a}}$$

$$\underbrace{\texttt{return}\ \underbrace{(\underbrace{\texttt{g}}_{::\ \texttt{a -> b}} \ \underbrace{\texttt{x}}_{::\ \texttt{a}})}_{::\ \texttt{b}}}_{::\ \texttt{IO b}}$$

Examples

...the following two versions of myAction are equivalent:

```
myAction :: IO String
myAction = do a <- getLine
              b <- getLine
              return $ a++b
```

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

Type and effect of myAction′ are similar but slightly different:

```
myAction' :: IO ()
myAction' =
 do a <- (++) <$> getLine <*> getLine
    putStrLn $ "Concatenation yields: " ++ a
```

Chapter 10.5.5

The Either Applicative

Homework

1. Make type constructor `(Either a)` an instance of `Applicative`.
2. Provide (most general) type information for the defining equations of the applicative operations `pure` and `(<*>)` of `(Either a)`.
3. Prove that `(Either a)` satisfies the applicative laws.

Chapter 10.5.6

The Map Applicative

The Map Applicative

...making the 1-ary type constructor ((-> d) an instance of the type constructor class Applicative:

```
instance Applicative ((->) d) where
  pure x  = (\_ -> x)
  g <*> h = \x -> g x (h x)
```

Lemma 10.5.6.1 (Applicative Laws for ((->) d))

The instance ((->) d) of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called map applicative.

The Map Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

```
instance Applicative ((->) d) where
```

$$\texttt{pure}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}\ \texttt{=}\ \underbrace{\texttt{(}\underbrace{\texttt{\textbackslash\_}}_{::\ \texttt{d}}\ \texttt{->}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}\texttt{)}}_{::\ \texttt{((->) d) a}}$$

$$\underbrace{\underbrace{\texttt{g}}_{::\ \texttt{((->) d) (a -> b)}}}_{::\ \texttt{d -> (a -> b)}}\ \texttt{<*>}\ \underbrace{\underbrace{\texttt{h}}_{::\ \texttt{((->) d) a}}}_{::\ \texttt{d -> a}}\ \texttt{=}\ \underbrace{\underbrace{\texttt{\textbackslash}\underbrace{\texttt{x}}_{::\ \texttt{d}}\ \texttt{->}\ \underbrace{\texttt{g}\ \underbrace{\texttt{x}}_{::\ \texttt{d}}\ \underbrace{\texttt{(h}\ \underbrace{\texttt{x}}_{::\ \texttt{d}}\texttt{)}}_{::\ \texttt{a}}}_{::\ \texttt{b}}}_{::\ \texttt{d -> b}}}_{::\ \texttt{((->) d) b}}$$

Examples

```
pure 3 "Hello"
 ->> (pure 3) "Hello"          (left-assoc. of expr.)
 ->> (\_ -> 3) "Hello"
 ->> 3
```

```
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
```

```
(+) <$> (+3) <*> (*100) $ 5 :: Int
 ->> (fmap (+) (+3)) <*> (*100) $ 5
 ->> ((+) . (+3)) <*> (*100) $ 5
 ->> (\x -> ((+) . (+3)) x ((*100) x)) $ 5
 ->> ((+) . (+3)) 5 ((*100) 5)
 ->> (+)((+3) 5) (5*100)
 ->> (+)(5+3) 500
 ->> (+) 8 500
 ->> (8+) 500
 ->> 8+500
 ->> 508 :: Int
```

Homework

...dealing with the map applicative.

Complete the stepwise evaluation of the below example:

```
(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
 ->> (fmap (\x y z -> [x,y,z]) (+3)) <*> (*2) <*> (/2) $ 5
 ->> ((\x y z -> [x,y,z]) . (+3)) <*> (*2) <*> (/2) $ 5
 ->> ...
 ->> [8.0,10.0,2.5]
```

Chapter 10.5.7

The Ziplist Applicative

The Ziplist Applicative (1)

...making the 1-ary type constructor ZipList an instance of the type constructor class Applicative:

```
newtype ZipList a = ZL [a]
 -- the newtype declaration is required since []
 -- can not be made a second time an instance
 -- of Applicative

instance Applicative ZipList where
 pure x           = ZL (repeat x)
 ZL gs <*> ZL xs = ZL (zipWith (\g x -> g x) gs xs)
```

Intuitively: <*> applies the first function to the first value, the second function to the second value, and so on.

The Ziplist Applicative (2)

Recall:

```
repeat :: a -> [a]
repeat x = x : repeat x -- generates stream [x,.x,..
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Lemma 10.5.7.1 (Applicative Laws for `ZipList`)

The instance `ZipList` of the type constructor class `Applicative` satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called ziplist applicative.

The Ziplist Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b

instance Applicative ZipList where

pure x = ZL (repeat x)
  :: a               :: a
            :: [a]
        :: ZipList a

    ZL gs       <*>   ZL xs
:: ZipList (a -> b)  :: ZipList a

  = ZL (zipWith (\g x -> g x)           gs          xs)
             :: (a->b,a) -> b      :: [(a->b)]   :: [a]
          :: ((a->b) -> a -> b)    :: [(a->b)]   :: [a]
                          :: [b]
                    :: ZipList b
```

Examples

```
getZipList $ (+) <$> ZL [1,2,3] <*> ZL [100,100,100]
 ->> getZipList $ (fmap (+) ZL [1,2,3]) <*> ZL [100,100,100]
 ->> getZipList $ ZL [(1+),(2+),(3+)] <*> ZL [100,100,100]
 ->> getZipList $ ZL [1+100,2+100,3+100]
 ->> getZipList $ ZL [101,102,103]
 ->> [101,102,103]

getZipList $ (+) <$> ZL [1,2,3] <*> ZL [100,100..]
 ->> getZipList $ (fmap (+) ZL [1,2,3]) <*> ZL [100,100,..]
 ->> getZipList $ ZL [(1+),(2+),(3+)] <*> ZL [100,100,..]
 ->> getZipList $ ZL [1+100,2+100,3+100]
 ->> [101,102,103]

getZipList $ max <$> ZL [1,2,3,4,5,3] <*> ZL [5,3,1,2]
 ->> ... ->> [5,3,3,4]

getZipList $ (,,) <$> ZL "dog" <*> ZL "cat" <*> ZL "rat"
 ->> ... ->> [('d','c','r'),('o','a','a'),('g','t','t')]
```

Chapter 10.6

Kinds of Types and Type Constructors

Kinds of Types and Type Constructors

Like values, also

- types
- type constructors

have types themselves, so-called

- kinds.

Kinds of Types

In GHCi, kinds of types (and type constructors) can be computed and displayed using the command "`:k`".

Examples:

```
ghci> :k Int
Int :: *
```

```
ghci> :k (Char,String)
(Char,String) :: *
```

```
ghci> :k [Float]
[Float] :: *
```

```
ghci> :k (Int -> Int)
(Int -> Int) :: *
```

where $*$ (read as "star" or as "type") indicates that the type is 'concrete' or 'final', i.e., a type accepting no type arguments.

Type Constructors

...take types as arguments to eventually produce concrete types.

Examples:

The type constructors `Maybe`, `Either`, and `Tree`

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data Tree a     = Leaf a  | Node a (Tree a) (Tree a)
```

produce for `a` and `b` chosen `Int` and `String`, respectively, the concrete types:

```
Maybe Int         :: *        -- a concrete type
Either Int String :: *        -- a concrete type
Tree Int          :: *        -- a concrete type
```

Kinds of Type Constructors

Like concrete types, type constructors have types, called kinds, too.

Examples:

```
ghci> :k Maybe
Maybe :: * -> *      -- a type constructor accepting
                     -- a concrete type as argument
                     -- and yielding a concrete type.

ghci> :k Either
Either :: * -> * -> *  -- a type constructor accepting
                       -- two concrete types as arguments
                       -- and yielding a concrete type.

ghci> :k Tree
Tree :: * -> *         -- like Maybe.

ghci> :k (->)
(->) :: * -> * -> *    -- like Either.
```

Kinds of Partially Evaluated Type Constructors

Like functions, also type constructors can be partially evaluated.

Examples:

```
ghci> :k Either
Either :: * -> * -> *   -- a type constructor accepting
                        -- two concrete types as arguments
                        -- and yielding a concrete type.

ghci> :k Either Int
Either Int :: * -> *    -- a type constructor accepting
                        -- one concrete type as argument
                        -- and yielding a concrete type.

ghci> :k Either Int Char
Either Int Char :: *     -- a concrete type.
```

Type Constructors as Functors

Recalling the definition of the type constructor class Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

it becomes obvious that only type constructors of kind

- `(* -> *)`

are eligible as possible instances of Functor.

Chapter 10.7

References, Further Reading

Chapter 10: Further Reading (1)

- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.1, The Functor Class; Chapter 18.5, Type Class Type Errors, Kinds of Types)
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 7, Making Our Own Types and Type Classes – The Functor Type Class; Chapter 11, Applicative Functors)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 10, Code Case Study: Parsing a Binary Data Format – Introducing Functors, Writing a Functor Instance for Parse, Using Functors for Parsing)

Chapter 10: Further Reading (2)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 11.1, Kategorien, Funktoren und Monaden)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 2.8.3, Type classes and inheritance)

Chapter 11

Monads

Monads: A Suisse Knife for Programming

Monadic programming is well suited for problems involving:

- Global state
  - Updating data during computation is often simpler than making all data dependencies explicit (state monad).
- Huge data structures
  - No need for replicating a data structure that is not needed otherwise.
- Exception and error handling
  - Maybe monad
- ...
- Side-effects, explicit sequencing and evaluation orders
  - Canonical scenario: Input/output operations (IO monad).

Chapter 11.1

Motivation

Motivation

....monads, a mundane approach for

- functional composition, for linking and sequencing functions!

The monad approach succeeds in

- linking and sequencing functions

whose types are incompatible and thus not amenable to

- ordinary functional composition `(.)`

Chapter 11.1.1

Functional Composition Reconsidered

Functional Composition

...means specifying the sequence of applications of functions:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)
```

If `f` and `g` are two functions of type:

```
f :: a -> b
g :: b -> c
```

then their composition is a function of type:

```
(g . f) :: a -> c
```

and applying `(g . f)` to some argument `x` means: Applying `f` to `x` first, applying second `g` to the result of `f` for `x`:

```
(g . f) x = g (f x) = let f_result = f x
                          g_result = g f_result
                      in g_result
```

R2L, L2R Sequencing of Function Applications

Sequencing from right to left (R2L):

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f v)
        ( = let f_result = f x
                g_result = g f_result
                in g_result  )
```

...enables R2L application sequences of the form:

```
(k . (... . (h . (g . f))...))
```

Sequencing from left to right (L2R):

```
(;) :: (a -> b) -> (b -> c) -> (a -> c)
(f ; g) = (g . f)
```

...enables L2R application sequences of the form:

```
((...((f ; g) ; h) ; ...) ; k)
```

R2L, L2R Sequencing: Two Derived Variants

The L2R sequencing variant (suggested by (;)):

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x
```

...enables L2R application sequences of the form:

```
(...(((x >>; f) >>; g) >>; h) >>; ... k)
  = x >>; f >>; g >>; h >>; ... k
```

The R2L sequencing variant (suggested by (.)):

```
(.<<) :: a -> (a -> b) -> b
f .<< x = f x                       -- Note: (.<<) = ($)
     ( = f $ x )
```

...enables R2L application sequences of the form:

```
(k ... .<< (h .<< (g .<< (f .<< x)))...)
 = k ... .<< h .<< g .<< f .<< x
```

Putting things together: It's all on Notation

...right-to-left (R2L) sequencing:

```
   (g . f) x      -- canonical sequencing notation
 = g (f x)
 = let f_result = f x
       g_result = g f_result
   in g_result
 =  g .<< f .<< x           -- notational variant
```

...left-to-right (L2R) sequencing:

```
    (f ; g) x              -- derived not. variant
  = (g . f) x
  = g (f x)
  = let f_result = f x
        g_result = g f_result
    in g_result
  = x >>; f >>; g         -- convenient not. variant
```

One more derived Sequencing Operation

...for left-to-right (L2R) sequencing:

```
(>;) :: a -> b -> b
x >; y  =  x >>; \_ -> y
     (  =  y )
```

(>;) enables L2R application sequences of the form:

```
(...(((x >; u) >; v) >; w) >; ... z)
  = x >; u >; v >; w >; ... z
  ->> z
```

which seems quite useless (and a notational overkill for just saying 'forget and drop the first argument') (but not so its monadic counterpart (>>)!)

Keep in mind

The monadic sequencing operations:

```
(>>=) :: m a -> (a -> m b) -> m b
(>>)  :: m a -> m b -> m b
```

...are the counterparts of:

```
(>>;) :: a -> (a -> b) -> b
(>;)  :: a -> b -> b
```

On Commonalities and Differences (1)

...of the monadic sequencing operations:

```
(>>=) :: m a -> (a -> m b) -> m b
c >>= f = ...                        -- Needs an m-specific
                                     -- implementation
(>>)  :: m a -> m b -> m b
c >> k = c >>= \_ -> k
```

...and their counterparts:

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x                        -- One implementation
                                     -- fits all types
(>;)  :: a -> b -> b
x >; y = x >>; \_-> y
```

On Commonalities and Differerences (2)

```
(>>;) :: a -> (a -> b) -> b
```

$$\underbrace{\texttt{x}}_{\texttt{:: a}}\ \ \texttt{>>;}\ \ \underbrace{\texttt{f}}_{\texttt{:: a -> b}}\ \ \texttt{=}\ \ \underbrace{\texttt{f x}}_{\texttt{:: b}}$$

```
(>;)  :: a -> b -> b
```

$$\underbrace{\underbrace{\texttt{x}}_{\texttt{:: a}}\ \ \texttt{>;}\ \ \underbrace{\texttt{y}}_{\texttt{:: b}}}_{\texttt{:: b}}\ \ \texttt{=}\ \ \underbrace{\underbrace{\texttt{x}}_{\texttt{:: a}}\ \ \texttt{>>;}\ \ \underbrace{\texttt{\textbackslash\_-> y}}_{\texttt{:: a -> b}}}_{\texttt{:: b}}$$

```
(>>=) :: m a -> (a -> m b) -> m b
```

$$\underbrace{\texttt{c}}_{\texttt{:: m a}}\ \ \texttt{>>=}\ \ \underbrace{\texttt{f}}_{\texttt{:: a -> m b}}\ \ \texttt{=}\ \ \underbrace{\texttt{...}}_{\texttt{:: m b}}$$

```
(>>)  :: m a -> m b -> m b
```

$$\underbrace{\underbrace{\texttt{c}}_{\texttt{:: m a}}\ \ \texttt{>>}\ \ \underbrace{\texttt{k}}_{\texttt{:: m b}}}_{\texttt{:: m b}}\ \ \texttt{=}\ \ \underbrace{\underbrace{\texttt{c}}_{\texttt{:: m a}}\ \ \texttt{>>=}\ \ \underbrace{\texttt{\textbackslash\_ -> k}}_{\texttt{:: a -> m b}}}_{\texttt{:: m b}}$$

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
Part IV
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
11.1
11.1.1
11.1.2
11.1.3
11.1.4

Note

...`(>>=)` is of type `(m a -> (a -> m b) -> m b)`, not of type `(m a -> (m a -> m b) -> m b)`!

A sequencing operation `(>>>=)`:

```
(>>>=) :: m a -> (m a -> m b) -> m b
c >>>= f = f c
```

could be implemented once and for all fitting all types just as the implementation of `(>>;)` fits all types:

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x
```

Often, however, we are lacking functions of type `(m a -> m b)` but have functions of type `(a -> m b)` instead.

Chapter 11.1.2

Example: Debug Information

Objective

...enhance two functions `f :: a -> b`, `g :: b -> c` such that they collect and output debug information during computation:

- `type Debug_Info = String`

To this end, replace `f` and `g` by functions `f'` and `g'`, which are as `f` and `g` but yield additionally to the results of `f` and `g` a piece of debug information:

```
f' :: a -> (b,Debug_Info)
g' :: b -> (c,Debug_Info)
```

Note: `f'` and `g'` can not be linked and sequenced using `(.)` since their argument and result types are incompatible and do not fit to each other.

Ad hoc Sequencing

...to overcome this problem, we could define a new function h whose implementation realizes the linking g and f, i.e., of sequentially composing them:

```
-- ‘h = f link g’ w/ the meaning: first f, then g
h :: a -> c
h x = let (f_result,f′_info) = f′ x
          (g_result,g′_info) = g′ f_result
      in (g_result,g′_info ++ f′_info)
```

Though working this were impractical as it continuously required implementing new functions (like h) which realize the sequencing of a pair of functions (like f′ and g′).

A new Sequencing Operator link_dbg

...more conveniently sequencing could be handled by introducing a function link_dbg for linking functions like f′ and g′:

```
link_dbg :: (a,Debug_Info) -> (a -> (b,Debug_Info))
                                    -> (b,Debug_Info)
link_dbg (x,s) g = let (g_result,g_info) = g x
                   in (g_result,s ++ g_info)
```

Note, link_dbg allows us to sequence f′ and g′ comfortably:

f′ x ‘link_dbg‘ g′ (= h x)

Example: Sequencing with `link_dbg`

Let:

```
f :: a -> b          f' :: a -> (b,Debug_Info)
f x =...             f' x = (f x, "f called, ")

g :: b -> c          g' :: b -> (c,Debug_Info)
g y =...             g' y = (g y, "g called, ")
```

Then:

```
f' x `link_dbg` g' `link_dbg` (\z -> (z, "done."))
 ->> (f x,"f called, ") `link_dbg` g' `link_dbg`
                                      (\z -> (z, "done."))
 ->> (g (f x),"f called, g called, ") `link_dbg`
                                      (\z -> (z, "done."))
 ->> (g (f x),"f called, g called, done.")
```

Chapter 11.1.3

Example: Random Numbers

Objective

The library `Data.Random` provides a function

```
random :: StdGen -> (a,StdGen)
```

for computing (pseudo) random numbers.

All functions `f :: a-> b` can use random numbers, if they can (additionally) manage a value of type `StdGen` to be used by the next function to generate a random number:

```
f :: a -> StdGen -> (b,StdGen)
g :: b -> StdGen -> (c,StdGen)
```

Note, `f` and `g` can not be sequenced using ordinary functional composition `(.)`.

Ad hoc Sequencing

...similarly to the 'debug' example, we could define a new function h, whose implementation realizes the sequential composition:

```
-- 'k = f link g' w/ the meaning: first f, then g
k :: a -> StdGen -> (c,StdGen)
k x gen = let (f_result,f_gen) = f x gen
              result = g f_result f_gen
          in result
```

Again, this works but were impractical as it continuously required implementing new functions (like k) which realize the sequencing of a pair of functions (like f and g).

A new Sequencing Operator link_rdm

...more conveniently sequencing could be handled by introducing a function `link_rdm` for linking functions like `f` and `g`:

```
link_rdm :: (StdGen -> (a,StdGen)) ->
                     (a -> StdGen -> (b,StdGen)) ->
                                      StdGen -> (b,StdGen)
link_rdm :: f g gen = let (y,f_gen) = f gen
                          (z,g_gen) = g y f_gen
                          result    = (z,g_gen)
                      in result
```

Note, `link_rdm` allow us to sequence `f` and `g` conveniently:

```
f x `link_rdm` g  ( =  k x )
```

Example: Sequencing with link_rdm

Let:
```
seed = ... :: StdGen
new_StdGen :: StdGen -> StdGen
new_StdGen gen = ...
f :: String -> StdGen -> (Int,StdGen)
f s gen = (length s,new_StdGen gen)
g :: Int -> StdGen -> (Bool,StdGen)
g n gen = (mod n 2 == 0,new_StdGen gen)
```

Then:
```
(f "Fun" `link_rdm` g) seed
 ->> let (m,f_gen) = (f "Fun") seed
         (b,g_gen) = g m f_gen
         result = (b,g_gen)
     in result
 ->> let (m,f_gen) = (length "Fun",new_StdGen seed)
         (b,g_gen) = (mod m 2 == 0,new_StdGen f_gen)
     in (b,g_gen)
 ->> let (m,f_gen) = (3,new_StdGen seed)
         (b,g_gen) = (mod 3 2 == 0,new_StdGen f_gen)
     in (b,g_gen)
 ->> (False,new_StdGen (new_StdGen seed))
```

Chapter 11.1.4

Findings, Looking ahead

Finding

...the examples of the two case studies enjoy a

- common structure.

This common structure can be encapsulated in a new

- type constructor class.

This type class is the type (constructor) class

- `Monad`.

Outlook: The Type Constructor Class Monad

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b   -- link
 return :: a -> m a     -- Value ‘lifting:’ Make an
                        -- (m a)-value; unit wrt (>>=)
 ...
```

...defining debug information and random numbers as new types allows to make them monads, i.e., instances of Monad:

```
newtype Dbg a = D (a,String)
newtype Rdm a = R (StdGen -> (a,StdGen))
```

such that:

```
(>>=)  :: Monad Dbg => Dbg a -> (a -> Dbg b) -> Dbg b
return :: Monad Dbg => a -> Dbg a

(>>=)  :: Monad Rdm => Rdm a -> (a -> Rdm b) -> Rdm b
return :: Monad Rdm => a -> Rdm a
```

Outlook: The Instance Declarations

...for 1) the type constructor Debug:

```
newtype Dbg a  = D (a,String)

instance Monad Dbg where
 (D (x,s)) >>= k = let D (x',s') = k (x,s)
                   in D (x',s ++ s')
 return x        = D (x,"")
```

...for 2) the type constructor Random:

```
newtype Rdm a = R (StdGen -> (a,StdGen))

instance Monad Rdm where
 (R f) >>= k = R $ \gen -> (let (x,gen') = f gen
                                (R b)    = k x
                            in b gen')
 return x    = R $ \gen -> (x,gen)
```

Chapter 11.1.5

Excursus on Functional Composition

Functors, Applicatives, Monads – Intuition (1)

...note the similarity of the signature patterns:

```
($) :: (a -> b) -> a -> b
g $ x = g x

fmap :: (Functor f) => (a -> b) -> f a -> f b
fmap g c = ...

(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
(<*>) k c = ...

(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>=) c k = ...

(.) :: (b -> c) -> (a -> b ) -> (a -> c)
(f . g) x = f (g x)
```

Functors, Applicatives, Monads – Intuition (2)

...in more detail with added type information:

```
($) :: (a -> b) -> a -> b
   g    $ x = g x
:: a->b  :: a  :: b

fmap :: (Functor f) => (a -> b) -> f a -> f b
fmap g       c   =  ...        -- w/ ... specific for f
  :: a->b  :: f a  :: f b

(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
(<*>) k          c    = ...      -- w/ ... specific for f
  :: f (a->b)  :: f a  :: f b

(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>=) c         k        = ...    -- w/ ... specific for m
    :: m a  :: a->m b)  :m b)

(.) :: (b -> c) -> (a -> b ) -> (a -> c)
(f . g) x = f (g x)
      :: a   :: c
```

Composing Functions: (.) and (;) (1)

...by default, function composition (or sequencing) in Haskell is from "right to left," just as in mathematics:

```
(.) :: (b -> c) -> (a -> b ) -> (a -> c)
(f . g) x = f (g x)
```

$\leadsto$ First g is applied, then f (application is "right to left!")

We complement "right to left" function compositition (.) with "left to right" function compositition (;):

```
(;) :: (a -> b) -> (b -> c ) -> (a -> c)
(f ; g) x = g (f x)
-- equivalently pointfree:
(f ; g) = g . f
```

$\leadsto$ First f is applied, then g (application is "left to right!")

Composing Functions: (.) and (;) (2)

Sequencing w/ (.): Functions are taken from "right to left:"

```
(fn . ... . f3 . f2 . f1 . f) x
  ->> (fn . ... . f3 . f2 . f1) (f x)
  ->> (fn . ... . f3 . f2) (f1 (f x))
  ->> (fn . ... . f3) (f2 (f1 (f x)))
  ->> ...
  ->>  fn ( ... ( f3 ( f2 ( f1 x)))...)
```

Sequencing w/ (;): Functions are taken from "left to right:"

```
(f ; f1 ; f2 ; f3 ; ... ; fn) x
  ->> (f1; f2 ; f3 ; ... ; fn) (f x)
  ->> (f2 ; f3 ; ... ; fn) (f1 (f x))
  ->> (f3 ; ... ; fn) (f2 (f1 (f x)))
  ->> ...
  ->>  fn ( ... ( f3 ( f2 ( f1 x)))...)
```

Relationship of `(.)` and `(;)` (1)

If `f`, `f1`, `f2`, `f3`,..., `fn` are functions and `x` a value of fitting types we have the following equalities:

```
(((fn  . ... . f3) . f2) . f1) . f =
     f ; (f1 ; (f2 ; (f3 ; ... ; fn)))

((((fn . ... . f3) . f2) . f1) . f) x =
      (f ; (f1 ; (f2 ; (f3 ; ... ; fn)))) x
```

Relationship (.) and (;) (2)

Both (.) and (;) are associative. Hence, parentheses can be dropped yielding:

```
fn . ... . f3 . f2 . f1 . f =
      f ; f1 ; f2 ; f3 ; ... ; fn

(fn . ... . f3 . f2 . f1 . f) x =
       (f ; f1 ; f2 ; f3 ; ... ; fn) x
```

Note:

- Both (.) and (;) specify explicitly the order, in which the functions are to be applied!
- This holds for monadic composition (>>=), too.
- Specifying sequencing precisely and explicitly is thus not a feature which is unique for monadic composition.

Sequencing for Monadic and Non-M. Types

In analogy to the monadic sequencing operator `(>>=)` for monads:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
     c    >>= k = ... :: m b
  (dc x)  >>= k = k x :: m b
    -- with dc some data constructor of type constructor m,
    -- and with x some value of type a, i.e, x :: a
```

...we introduce a sequencing operator `(>>;)` inspired by `(>>=)` and `(;)` for non-monadic types:

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x :: b
```

Sequencing Functions w/ (;) and (>>;)

The operators (;) and (>>;) are closely related:

```
(f ; f1 ; f2 ; f3 ; ... ; fn) x =
   x >>; f >>; f1 >>; f2 >>; f3 >>; ... >>; fn
```

(;): function application left to right but argument on the right.

```
(f ; f1 ; f2 ; f3 ; ... ; fn) x
  ->> (f1; f2 ; f3 ; ... ; fn) (f x)
  ->> (f2 ; f3 ; ... ; fn) (f1 (f x))
  ->>  ...
  ->>  fn ( ... ( f3 ( f2 ( f1 x)))...)
```

(>>;): function application left to right and argument on the left!

```
x >>; f >>; f1 >>; f2 >>; f3 >>; ... >>; fn
  ->> (f x) >>; f1 >>; f2 >>; f3 >>; ... >>; fn
  ->> (f1 (f x)) >>; f2 >>; f3 >>; ... >>; fn
  ->>  ...
  ->>  fn ( ... ( f3 ( f2 ( f1 x)))...)
```

Non-Monadic Function Sequencing: (>>;) (1)

$$\underbrace{\texttt{x}}_{\texttt{:: a}}\ \texttt{>>;}\ \underbrace{\texttt{f}}_{\texttt{:: a -> b}}\ \texttt{>>;}\ \underbrace{\texttt{f1}}_{\texttt{:: b -> c}}\ \texttt{>>;}\ \underbrace{\texttt{f2}}_{\texttt{:: c -> d}}\ \texttt{>>;}\ \underbrace{\texttt{f3}}_{\texttt{:: d -> e}}\ \texttt{>>;}\ \underbrace{\texttt{f4}}_{\texttt{:: e -> g}}\ \ \texttt{:: g}$$

$$\underbrace{\underbrace{\texttt{id x}}_{\texttt{:: a}}\ \texttt{>>;}\ \underbrace{\texttt{f}}_{\texttt{:: a -> b}}}_{\texttt{x1}}$$

$$\underbrace{\underbrace{\texttt{x1}}_{\texttt{:: b}}\ \texttt{>>;}\ \underbrace{\texttt{f1}}_{\texttt{:: b -> c}}}_{\texttt{x2}}$$

$$\underbrace{\underbrace{\texttt{x2}}_{\texttt{:: c}}\ \texttt{>>;}\ \underbrace{\texttt{f2}}_{\texttt{:: c -> d}}}_{\texttt{x3}}$$

$$\underbrace{\underbrace{\texttt{x3}}_{\texttt{:: d}}\ \texttt{>>;}\ \underbrace{\texttt{f3}}_{\texttt{:: d -> e}}}_{\texttt{x4}}$$

$$\underbrace{\underbrace{\texttt{x4}}_{\texttt{:: e}}\ \texttt{>>;}\ \underbrace{\texttt{f4}}_{\texttt{:: e -> g}}}_{\texttt{x5}}$$

$$\underbrace{\texttt{x5}}_{\texttt{:: g}}$$

Non-Monadic Function Sequencing: (>>;)(2)

The same but (most) types dropped and parentheses added for clarity:

$$\mathtt{(((((}\underbrace{\mathtt{x}}_{\mathtt{::\ a}}\ \mathtt{>>;}\ \underbrace{\mathtt{f)}}_{\mathtt{::\ a\ ->\ b}}\ \mathtt{>>;}\ \underbrace{\mathtt{f1)}}_{\mathtt{::\ b\ ->\ c}}\ \mathtt{>>;}\ \underbrace{\mathtt{f2)}}_{\mathtt{::\ c\ ->\ d}}\ \mathtt{>>;}\ \underbrace{\mathtt{f3)}}_{\mathtt{::\ d\ ->\ e}}\ \mathtt{>>;}\ \underbrace{\mathtt{f4)}}_{\mathtt{::\ e\ ->\ g}}\ \mathtt{::\ g}$$

```
(((((x >>; f) >>; f1) >>; f2) >>; f3) >>; f4)
   ->> ((((x1 >>; f1) >>; f2) >>; f3) >>; f4)
             ->> (((x2 >>; f2) >>; f3) >>; f4)
                     ->> ((x3 >>; f3) >>; f4)
                             ->> (x4 >>; f4)
                                    ->> x5 :: g
```

Non-Monadic Function Sequencing: (>>;)(3)

The same but (most) types and parentheses dropped:

$$\underbrace{\texttt{x}}_{\texttt{:: a}}\ \texttt{>>;}\ \underbrace{\texttt{f}}_{\texttt{:: a -> b}}\ \texttt{>>;}\ \underbrace{\texttt{f1}}_{\texttt{:: b -> c}}\ \texttt{>>;}\ \underbrace{\texttt{f2}}_{\texttt{:: c -> d}}\ \texttt{>>;}\ \underbrace{\texttt{f3}}_{\texttt{:: d -> e}}\ \texttt{>>;}\ \underbrace{\texttt{f4}}_{\texttt{:: e -> g}}\ \texttt{:: g}$$

```
x   >>;  f   >>;  f1  >>;  f2  >>;  f3  >>;  f4
    ->>  x1  >>;  f1  >>;  f2  >>;  f3  >>;  f4
             ->>  x2  >>;  f2  >>;  f3  >>;  f4
                      ->>  x3  >>;  f3  >>;  f4
                               ->>  x4  >>;  f4
                                        ->>  x5 :: g
```

Note: The operators (>>;) are applied from left to right and the argument is forwarded from left to right, too. This gets lost if (>>;) is used as prefix operator (cf. next slide).

Non-Monadic Function Sequencing: (>>;) (4)

Infix usage of (>>;):

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\texttt{x >>; f}}_{\texttt{x1}}\texttt{ >>; f1}}_{\texttt{x2}}\texttt{ >>; f2}}_{\texttt{x3}}\texttt{ >>; f3}}_{\texttt{x4}}\texttt{ >>; f4}}_{\texttt{x5}}\texttt{ :: g}$$

...vs. prefix usage of (>>;):

$$\underbrace{\texttt{(>>;) }\underbrace{\texttt{((>>;) }\underbrace{\texttt{((>>;) }\underbrace{\texttt{((>>;) }\underbrace{\texttt{((>>;) x f)}}_{\texttt{x1}}\texttt{ f1)}}_{\texttt{x2}}\texttt{ f2)}}_{\texttt{x3}}\texttt{ f3)}}_{\texttt{x4}}\texttt{ f4)}}_{\texttt{x5}}\texttt{ :: g}$$

Monadic Function Sequencing: (>>=) (1)

$$\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{f}}_{\texttt{:: a -> m b}} \;\texttt{>>=}\; \underbrace{\texttt{f1}}_{\texttt{:: b -> m c}} \;\texttt{>>=}\; \underbrace{\texttt{f2}}_{\texttt{:: c -> m d}} \;\texttt{>>=}\; \underbrace{\texttt{f3}}_{\texttt{:: d -> m e}} \;\texttt{>>=}\; \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}} \;\texttt{:: m g}$$

$$\underbrace{\underbrace{\texttt{return}\;\underbrace{\texttt{c0}}_{\texttt{:: a}}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{f}}_{\texttt{:: a -> m b}}}_{\texttt{c1}}$$

$$\underbrace{\underbrace{\texttt{c1}}_{\texttt{:: m b}} \;\texttt{>>=}\; \underbrace{\texttt{f1}}_{\texttt{:: b -> m c}}}_{\texttt{c2}}$$

$$\underbrace{\underbrace{\texttt{c2}}_{\texttt{:: m c}} \;\texttt{>>=}\; \underbrace{\texttt{f2}}_{\texttt{:: c -> m d}}}_{\texttt{c3}}$$

$$\underbrace{\underbrace{\texttt{c3}}_{\texttt{:: m d}} \;\texttt{>>=}\; \underbrace{\texttt{f3}}_{\texttt{:: d -> m e}}}_{\texttt{c4}}$$

$$\underbrace{\underbrace{\texttt{c4}}_{\texttt{:: m e}} \;\texttt{>>=}\; \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}}}_{\texttt{c5}}$$

$$\underbrace{\texttt{c5}}_{\texttt{:: m g}}$$

Monadic Function Sequencing:: (>>=) (2)

The same but (most) types dropped and parentheses added for clarity:

$$\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{f}}_{\texttt{:: a -> m b}} \;\texttt{>>=}\; \underbrace{\texttt{f1}}_{\texttt{:: b -> m c}} \;\texttt{>>=}\; \underbrace{\texttt{f2}}_{\texttt{:: c -> m d}} \;\texttt{>>=}\; \underbrace{\texttt{f3}}_{\texttt{:: d -> m e}} \;\texttt{>>=}\; \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}} \;\texttt{:: m g}$$

```
(((((c >>= f) >>= f1) >>= f2) >>= f3) >>= f4)
   ->> ((((c1 >>= f1) >>= f2) >>= f3) >>= f4)
          ->> (((c2 >>= f2) >>= f3) >>= f4)
                 ->> ((c3 >>= f3) >>= f4)
                        ->> (c4 >>= f4)
                               ->> c5 :: m g
```

Monadic Function Sequencing:: (>>=) (3)

The same but (almost all) types and parentheses dropped:

$$\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{f}}_{\texttt{:: a -> m b}} \;\texttt{>>=}\; \underbrace{\texttt{f1}}_{\texttt{:: b -> m c}} \;\texttt{>>=}\; \underbrace{\texttt{f2}}_{\texttt{:: c -> m d}} \;\texttt{>>=}\; \underbrace{\texttt{f3}}_{\texttt{:: d -> m e}} \;\texttt{>>=}\; \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}} \;\texttt{:: m g}$$

```
c   >>=  f   >>=  f1  >>=  f2  >>=  f3  >>=  f4
    ->>  c1  >>=  f1  >>=  f2  >>=  f3  >>=  f4
              ->>  c2  >>=  f2  >>=  f3  >>=  f4
                        ->>  c3  >>=  f3  >>=  f4
                                 ->>  c4  >>=  f4
                                          ->>  c5 :: m g
```

Note: The operators (>>=) are applied from left to right and the argument is forwarded from left to right, too. This gets lost if (>>=) is used as prefix operator (cf. next slide).

Monadic Function Sequencing: (>>=) (4)

Infix usage of (>>=):

```
c  >>=  f  >>=  f1  >>=  f2  >>=  f3  >>=  f4  :: m g
```

c1 c2 c3 c4 c5

...vs. prefix usage of (>>=):

```
(>>=) ((>>=) ((>>=) ((>>=) ((>>=) c f) f1) f2) f3) f4) :: m g
```

c1 c2 c3 c4 c5

Monadic Function Sequencing via do-Not. (1)

$$\underbrace{\texttt{v}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{\texttt{:: a -> m b}}\ \texttt{>>=}\ \underbrace{\texttt{f1}}_{\texttt{:: b -> m c}}\ \texttt{>>=}\ \underbrace{\texttt{f2}}_{\texttt{:: c -> m d}}\ \texttt{>>=}\ \underbrace{\texttt{f3}}_{\texttt{:: d -> m e}}\ \texttt{>>=}\ \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}}\ \texttt{:: m g}$$

$$\begin{array}{lcll}
\texttt{do}\ \underbrace{\texttt{v0}'}_{\texttt{:: a}} & \texttt{<-} & \underbrace{\texttt{return}\ \underbrace{\texttt{v0}}_{\texttt{:: a}}}_{\texttt{:: m a}} & \texttt{-- Note: return v0 ->> v} \\
\quad \underbrace{\texttt{v1}}_{\texttt{:: b}} & \texttt{<-} & \underbrace{\texttt{f v0}'}_{\texttt{:: m b}} & \\
\quad \underbrace{\texttt{v2}}_{\texttt{:: c}} & \texttt{<-} & \underbrace{\texttt{f1 v1}}_{\texttt{:: m c}} & \\
\quad \underbrace{\texttt{v3}}_{\texttt{:: d}} & \texttt{<-} & \underbrace{\texttt{f2 v2}}_{\texttt{:: m d}} & \\
\quad \underbrace{\texttt{v4}}_{\texttt{:: e}} & \texttt{<-} & \underbrace{\texttt{f3 v3}}_{\texttt{:: m e}} & \\
\quad \underbrace{\texttt{v5}}_{\texttt{:: g}} & \texttt{<-} & \underbrace{\texttt{f4 v4}}_{\texttt{:: m g}} & \\
\quad \underbrace{\texttt{return v5}}_{\texttt{:: m g}} & & &
\end{array}$$

Monadic Function Sequencing via do-Not. (2)

The expression

$$\texttt{(((((}\underbrace{\texttt{v}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{f)}}_{\texttt{:: a -> m b}}\ \texttt{>>=}\ \underbrace{\texttt{f1)}}_{\texttt{:: b -> m c}}\ \texttt{>>=}\ \underbrace{\texttt{f2)}}_{\texttt{:: c -> m d}}\ \texttt{>>=}\ \underbrace{\texttt{f3)}}_{\texttt{:: d -> m e}}\ \texttt{>>=}\ \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}}\ \texttt{) :: m g}$$

...in standard notation using (>>=) and parentheses for order specification can equivalently be written using the syntactic sugar of the do-notation

```
do v0' :: a <- return v0 :: m a -- Note:
   v1  :: b <- f  v0'    :: m b    -- return v0 ->> v
   v2  :: c <- f1 v1     :: m c
   v3  :: d <- f2 v2     :: m d
   v4  :: e <- f3 v3     :: m e
   v5  :: g <- f4 v4     :: m g
   return v5 :: m g
```

...with an implicit ordering specification by data dependencies.

Monadic Function Sequencing via do-Not. (3)

The same but (most) types dropped...

The expression

$$(((((\underbrace{\texttt{v}}_{\texttt{:: m a}} \texttt{ >>= } \underbrace{\texttt{f}}_{\texttt{:: a -> m b}}) \texttt{ >>= } \underbrace{\texttt{f1}}_{\texttt{:: b -> m c}}) \texttt{ >>= } \underbrace{\texttt{f2}}_{\texttt{:: c -> m d}}) \texttt{ >>= } \underbrace{\texttt{f3}}_{\texttt{:: d -> m e}}) \texttt{ >>= } \underbrace{\texttt{f4}}_{\texttt{:: e -> m g}}) \texttt{ :: m g}$$

...is equivalent to the do-expression:

```
do v0' <- return v0    -- Note: return v0 ->> v
   v1  <- f  v0'
   v2  <- f1 v1
   v3  <- f2 v2
   v4  <- f3 v3
   v5  <- f4 v4
   return v5
```

Compare: Monadic vs. Non-M. Operations (1)

A non-monadic application example:

```
"Functional Programming" >>; length >>; odd >>; f
```

where

```
f :: Bool -> Char
f True  = 'H'   -- reminding to High
f False = 'L'   -- reminding to Low
```

...stepwise evaluated:

$$\underbrace{\underbrace{\underbrace{\texttt{"Functional Programming" >>; length}}_{\texttt{22}}\ \texttt{>>; odd}}_{\texttt{False}}\ \texttt{>>; f}}_{\texttt{'L'}}$$

Compare: Monadic vs. Non-M. Operations (2)

...and its monadic counterpart:

```
Id "Functional Programming" >>= length_m >>= odd_m >>= f_m
```

where

```
length_m :: String -> Id Int
length_m s = Id (length s)
odd_m :: Int -> Id Bool
odd_m n = Id (odd n)
f_m :: Bool -> Id Char
f_m b = Id (f b)
```

...stepwise evaluated:

$$\underbrace{\underbrace{\underbrace{\texttt{Id "Functional Programming" >>= length\_m}}_{\texttt{Id 22}}\ \texttt{>>= odd\_m}}_{\texttt{Id False}}\ \texttt{>>= f\_m}}_{\texttt{Id `L`}}$$

Compare: Monadic vs. Non-M. Operations (3)

Monadic operations

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
c >>= k = k x :: m b
(dc x)  -- w/ dc a data constructor of type constructor m, and w/ x a value of type a, i.e., x :: a
return :: (Monad m) => a -> m a
return v = m v :: m a
fail :: (Monad m) => String -> m a
fail s = error s :: m a
(>>) :: (Monad m) => m a -> m b -> m b
c >> k = c >> \_ -> k :: m b
```

...and their non-monadic counterparts:

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x :: b
id :: a -> a
id x -> x :: a
fail :: String -> a
fail s = error s :: a
(>;) :: a -> b -> b
x >; y = x >>; \_ -> y :: b    -- i.e.: x >; y = y :: b
```

Why Introducing Monads at All? (1)

...generality, flexibility, and re-use!

Note, just staying with

```
(>>;) :: a -> (a -> b) -> b
v >>; f = f v
```

means to stay

- with only one implementation of `(>>;)` for all types `a` and `b`
- which must be used and work for all types `a` and `b`
- which thus can not be particularly "type specific" since nothing can be assumed about `a` and `b` by the implementation of `(>>;)`

Why Introducing Monads at All? (2)

Note, (>>;) does not allow to cope with the debug-example.

```
f :: String -> Int       g :: Int -> Bool
f = length               g = odd

(g . f)   =  f ; g -- composition of f and g works!
(g . f) s = (f ; g) s = g(f(s)) -- works for all values
          = s >>; f >>; g       -- s of type String!
```

While composition works fine for f & g, it does not for f′ & g′:

```
f′ :: String -> (Int,String) g′ :: Int -> (Bool,String)
f′ s = (f s,"f called, ")    g′ n = (g n,"g called, ")

(g′ . f′) = f′ ; g′ -- does not work: types of g’
                    -- and f’ do not fit!
(g′ . f′) s = (f′ ; g′) s = g′(f′(s)) -- does not work:
              = s >>; f′ >>; g′ -- type-specific implemen-
                                -- tations of (>>;), (>>;)
                                -- are required!
```

Why Introducing Monads at All? (3)

Introduce a new data type `Debug a`:

```
newtype Debug a  = D (a,String)
```

Make the constructor `Debug` an instance of class `Monad`:

```
instance Monad Debug where
 (D (v,s)) >>= f = let D (v′,s′) =
                          f (v,s) in D (v′,s++s′)
 return x        = D (x,"")
```

Note that `Debug Int` and `Debug Bool` are both instances of type `Debug a`. This allows us to switch from `f′`, `g′` to `f_m`, `g_m`:

```
f_m :: String -> Debug Int      g_m :: Int -> Debug Bool
f_m s = D (f s,"f called. ")  g_m n = D (g n,"g called. ")
```

```
D (s,t) >>= f_m >>= g_m -- works for all values s, t of
                        -- type String!
```

Hence, we got the desired type-awareness of `(>>=)` with just one instance declaration!

Why Introducing Monads at All? (4)

In fact, introducing the type constructor class Monad

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b
 return :: a -> m a
 (>>)   :: m a -> m b -> m b
 fail   :: String -> m a
```

allows as many implementations of (>>=) for a type as needed. It only requires to hide the type behind a distinct new type constructor to allow another implementation of (>>=) for it:

```
data Id a     = ...; instance Monad Id where...
data [] a     = ...; instance Monad [] where...
data Maybe a  = ...; instance Monad Maybe where...
data Tree a   = ...; instance Monad Tree where...
data IO a     = ...; instance Monad IO where...
...
data Id' a    = ...; instance Monad Id' where...
data Maybe' a = ...; instance Monad Maybe' where...
...
```

Why Introducing Monads at All? (5)

...where (the values of) the data types

```
data Id' a     = Id' a
data Maybe' a = Nothing' | Just' a
data List' a  = Empty' | Cons' a (List' a)
...
```

equal their "unprimed" counterparts but allow us to implement a different behaviour for `(>>=)` and the other monadic operations.

Why Introducing Monads at All? (6)

All in all, this also allows to interleave applications of (>>=) and (>>;) and to change the monad in the course of the computation, e.g., from Id to Id′):

```
id2id' :: Id a -> Id' a        id'2id :: Id' a -> Id a
id2id' (Id v) = Id' v         id'2id (Id' v) = Id v

s = Id "Fun" :: Id String

f , g   :: String -> Id String
f', g'  :: String -> Id' String
```

$$\underbrace{\texttt{s >>= f}}_{\texttt{mon.c.}}\ \overbrace{\underbrace{\texttt{>>;}}_{\texttt{ord.c.}}\ \texttt{id2id'}\ \underbrace{\texttt{>>= f'}}_{\texttt{mon.c.}}}^{\texttt{monad change: Id2Id'}}\ \underbrace{\texttt{>>=}}_{\texttt{mon.c.}}\ \overbrace{\texttt{g'}\ \underbrace{\texttt{>>;}}_{\texttt{ord.c.}}\ \texttt{id'2id}\ \underbrace{\texttt{>>= g}}_{\texttt{mon.c.}}}^{\texttt{monad change: Id'2Id}}$$

Last but not least (1)

If we had been prepared to change both domain and range of functions (instead of their range only), ordinary composition would have been sufficient for the debug-example:

While

```
f′ :: String -> (Int,String)     g′ :: Int -> (Bool,String)
f′ s = (f s,"f called, ")        g′ n = (g n,"g called, ")

(g′ . f′) = f′ ; g′ -- does not work: types of g′ and f′
                    -- do not fit!
(g′ . f′) s = (f′ ; g′) s = g′(f′(s)) -- does not work:
            = s >>; f′ >>; g′ -- type-specific implemen-
                              -- tations are required!
```

does not work, the following does work:

```
f" :: = (String,String) -> (Int,String)
f" (s,t) = (f s , t ++ "f called, ")
g" :: (Int,String) -> (Bool,String)
g" (n,t) = (g n , t ++ "g called, ")
(g" . f") s = (f" ; g") s = g"(f"(s)) = (s,"") >>; f" >>; g"
```

Last but not least (2)

Compare the monadic-free implementation of the debug-example....

```
f" :: = (String,String) -> (Int,String)  -- Note: Concatenation of
f" (s,t) = (f s , t ++ "f called, ")      -- Strings handled by
g" :: (Int,String) -> (Bool,String)       -- f" and g", not by (>>;)
g" (n,t) = (g n , t ++ "g called, ")

("Fun","") >>; f" >>; g"
->> (3,"f called, ") >>; g" ->> (True,"f called, g called, ")
```

...with its monadic counterpart:

```
newtype Debug a  = D (a,String)
instance Monad Debug where                      -- Note: Concatenation
 (D (v,s)) >>= f = let D (v′,s′) =             --  of Strings handled
                   f (v,s) in D (v',s ++ s')   -- by (>>=), not
 return x        = D (x,"")                     --  by fm and gm
fm :: String -> Debug Int     gm :: Int -> Debug Bool
fm s = D (f s,"f called, ")  gm n = D (g n,"g called, ")

D (s,"") >>= fm >>= gm
->> D (3,"f called, ") >>; gm ->> D (True,"f called, g called, ")
```

Quite similar, aren't they?

Chapter 11.2

The Type Constructor Class Monad

The Type Constructor Class Monad

Type Constructor Class Monad

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b
 return :: a -> m a
 (>>)   :: m a -> m b -> m b
 fail   :: String -> m a

 c >> k  =  c >>= \_ -> k
 fail s  =  error s
```

...monads are instances of the type constructor class Monad (hence 1-ary type constructors), which obey the monad laws:

Monad Laws

```
return x >>= f                  = f x                (ML1)
c >>= return                    = c                  (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g   (ML3)
```

Type Constructor Class Monad in more Detail

```
class Monad m where
 -- 'Primary' functions (relevant for every monad)
 return :: a -> m a               -- Value 'lifting:' Ma-
                                  -- king a monadic value
 (>>=) :: m a -> (a -> m b) -> m b    -- Sequencing
 -- 'Secondary' functions (relevant for some monads)
 fail :: String -> m a            -- Error handling
 (>>) :: m a -> m b -> m b        -- Simplified sequencing
 -- Default implementations
 fail s       =  error s          -- Failing computation:
  :: String        :: String      -- Outputting s as an
 :: m a          :: m a           -- error message
   c   >>  k    =    c   >>=  \_ -> k
 :: m a :: m b     :: m a     :: a -> m b
     :: m b                :: m b
```

The Monad Laws in more Detail

...with added type information:

$$\underbrace{\underbrace{\underbrace{\texttt{return}}_{\texttt{:: a -> m a}}\ \underbrace{\texttt{x}}_{\texttt{:: a}}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{\texttt{:: a -> m b}}}_{\texttt{:: m b}} \ \texttt{=}\ \underbrace{\underbrace{\texttt{f}}_{\texttt{:: a -> m b}}\ \underbrace{\texttt{x}}_{\texttt{:: a}}}_{\texttt{:: m b}} \qquad \texttt{(ML1)}$$

$$\underbrace{\underbrace{\texttt{c}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{return}}_{\texttt{:: a -> m a}}}_{\texttt{:: m a}} \ \texttt{=}\ \underbrace{\texttt{c}}_{\texttt{:: m a}} \qquad \texttt{(ML2)}$$

Homework

Provide (most general) type information for the monad law ML3:

```
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g   (ML3)
```

Note

...the monad laws require from (proper) monad instances:

- `return` passes its argument without any other effect, i.e., `return` is unit of `(>>=)` (see also function `pure` of class `Applicative`) (ML1, ML2).
- `(>>=)` is associative, i.e., sequencings given by `(>>=)` must not depend on how they are bracketed (ML3).

Proof obligation:

- It is a programmer obligation to prove that their instances of `Monad` satisfy the monad laws.

Note: Sequence operator `(>>=)`: Read as bind (Paul Hudak) or then (Simon Thompson). Sequence operator `(>>)`: Derived from `(>>=)`, read as sequence (Paul Hudak).

Associativity of (>>)

Lemma 11.2.1 (Associativity of (>>))

If (>>=) of some monad m is associative, then also the default implementation of (>>) is associative, i.e.:

```
c1 >> (c2 >> c3) = (c1 >> c2) >> c3
```

The Operator (>@>)

Note, the formulation of associativity for (>>=):

```
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g
```

...is less appealing than the one for (>>):

```
c1 >> (c2 >> c3) = (c1 >> c2) >> c3
```

The operator (>@>) derived from (>>=) and defined by:

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c)
                                         -> (a -> m c)
f >@> g = \x -> (f x) >>= g
```

...improves on this: For (>@>), the monad laws, especially the associativity requirement, become as natural and obvious as for (>>).

The Monad Laws in Terms of (>@>)

Lemma 11.2.2

If (>>=) and return of some monad m are associative and unit of (>>=), respectively, then we have:

```
return >@> f      = f                    (ML1')
f >@> return      = f                    (ML2')
(f >@> g) >@> h = f >@> (g >@> h)        (ML3')
```

Intuitively

- return is unit of (>@>) (ML1', ML2').
- (>@>) is associative (ML3').

A Law linking Classes Monad and Functor

...type constructors, which shall be proper instances of both Monad and Functor must satisfy law MFL:

```
fmap g xs  =  xs >>= return . g                    (MFL)
         ( =  do x <- xs; return (g x) )
```

Selected Utility Functions for Monads (1)

```
(=<<)        :: Monad m => (a -> m b) -> m a -> m b
f =<< x      =  x >>= f

sequence     :: Monad m => [m a] -> m [a]
sequence     =  foldr mcons (return [])
                  where mcons p q = do l  <- p
                                       ls <- q
                                       return (l:ls)

sequence_    :: Monad m => [m a] -> m ()
sequence_    =  foldr (>>) (return ())

mapM         :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as    =  sequence (map f as)

mapM_        :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as =  sequence_ (map f as)
```

Selected Utility Functions for Monads (2)

```
mapF        :: Monad m => (a -> b) -> m [a] -> m [b]
mapF f x    =  do v <- x; return (f v)
   -- equals map on lists, i.e., for picking [] as m

joinM       :: Monad m => m (m a) -> m a
joinM x     =  do v <- x; v
 -- equals concat on lists, i.e., for picking [] as m
```

...and many more (see e.g., library Monad).

Lemma 11.2.3

1. `mapF (f . g) = mapF . mapF g`
2. `joinM return = joinM . mapF return`
3. `joinM return = id`

Homework

1. Prove Lemma 11.2.3.
2. Do the functor and monad laws imply law FML? Provide a proof or a counter-example.
3. Provide (most general) type information for
   3.1 the defining equation of (>@>):
   ```
   (>@>) :: Monad m => (a -> m b) -> (b -> m c)
                                          -> (a -> m c)
   f >@> g = \x -> (f x) >>= g
   ```
   3.2 the statement of Lemma 11.2.1:
   ```
   c1 >> (c2 >> c3) = (c1 >> c2) >> c3
   ```
   3.3 the statements of Lemma 11.2.2:
   ```
   return >@> f     = f                    (ML1')
   f >@> return     = f                    (ML2')
   (f >@> g) >@> h = f >@> (g >@> h)       (ML3')
   ```

Chapter 11.3

Syntactic Sugar: The do-Notation

The do-Notation

...the monadic operations `(>>=)` and `(>>)` allow very much as functional composition `(.)`

- to specify the sequencing of (fitting) operations explicitly.

Both functional and monadic sequencing introduce

- an imperative flavour into functional programming.

Using the so-called

- `do`-notation

as syntactic sugar expresses this flavour for monadic sequencing in a syntactically more appealing and concise fashion.

Relating Monadic Operations and do-Notation

...four conversion rules allow the conversion of sequences of monadic operations composed of

- `(>>=)` and `(>>)`

into equivalent ('<=>') sequences of

- `do`-blocks

and vice versa.

Intuitively

Recall:

```
(>>=) :: m a -> (a -> m b) -> m b
(>>)  :: m a -> m b -> m b
```

Then:

$$\underbrace{\texttt{dc v}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{\texttt{:: (a -> m b)}}\ \texttt{->>}\ \underbrace{\texttt{f v}}_{\texttt{:: m b}}$$

$$\texttt{"<=> do}\ \underbrace{\texttt{x}}_{\texttt{:: a}}\ \texttt{<-}\ \underbrace{\texttt{dc v}}_{\texttt{:: m a}}\texttt{;}\ \underbrace{\texttt{y}}_{\texttt{:: b}}\ \texttt{<-}\ \underbrace{\texttt{f x}}_{\texttt{:: m b}}\texttt{;}\ \underbrace{\texttt{return y}}_{\texttt{:: m b}}\texttt{"}$$

$$\underbrace{\texttt{dc v}}_{\texttt{:: m a}}\ \texttt{>>}\ \underbrace{\texttt{dc' v'}}_{\texttt{:: m b}}\ \texttt{->>}\ \underbrace{\texttt{dc v}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{\_ -> dc' v'}}_{\texttt{:: (a -> m b)}}$$

$$\texttt{"<=> do}\ \underbrace{\texttt{\_}}_{\texttt{:: a}}\ \texttt{<-}\ \underbrace{\texttt{dc v}}_{\texttt{:: m a}}\texttt{;}\ \underbrace{\texttt{y}}_{\texttt{:: b}}\ \texttt{<-}\ \underbrace{\texttt{dc' v'}}_{\texttt{:: m b}}\texttt{;}\ \underbrace{\texttt{return y}}_{\texttt{:: m b}}\texttt{"}$$

with dc, dc′ some data constructors of type constructor m.

The Conversion Rules

```
(R1) do e <=> e

(R2) do e1;e2;...;en <=> e1 >>= \_ -> do e2;...;en
                     <=> e1 >> do e2;...;en

(R3) do let decl_list;e2;...;en <=> let decl_list
                                    in do e2;...;en

(R4) do pattern <- e1;e2;...;en <=>
         let ok pattern = do e2;...;en
             ok _       = fail "..."
             in e1 >>= ok
```

...and as a special case of the 'pattern' rule (R4):

```
(R4') do x <- e1;e2;...;en <=>
         e1 >>= \x -> do e2;...;en
```

Notes on the Conversion Rules

Intuitively

- (R2): If the return value of an operation is not needed, it can be moved to the front.
- (R3): A `let`-expression storing a value can be placed in front of the `do`-block.
- (R4): Return values that are bound to a pattern, require a supporting function that handles the pattern matching and the execution of the remaining operations, or that calls `fail`, if the pattern matching fails.

  Note: It is rule (R4) which necessitates `fail` as a monadic operation in `Monad`. Overwriting this operation allows a monad-specific exception and error handling.

Illustrating the do-Notation

...using the monad laws as example.

A) The monad laws using (>>=) and (>>):

```
return a >>= f              = f a                   (ML1)
c >>= return                = c                     (ML2)
c >>= (\x -> (f x) >>= g)   = (c >>= f) >>= g       (ML3)
```

B) The monad laws using do-notation:

```
do x <- return a; f x       =  f a                  (ML1)
do x <- c; return x         =  c                    (ML2)
do x <- c; y <- f x; g y    =
           do y <- (do x <- c; f x); g y            (ML3)
```

Semicolons vs. Linebreaks in do-Notation

B) do-notation in 'one' line (w/ ';', no linebreaks):

```
do x <- return a; f x      =  f a                        (ML1)
do x <- c; return x        =  c                          (ML2)
do x <- c; y <- f x; g y   =
            do y <- (do x <- c; f x); g y                (ML3)
```

C) do-notation in 'several' lines (w/ linebreaks, no ';'):

```
do x <- return a
   f x             =  f a                                (ML1)
do x <- c
   return x        =  c                                  (ML2)
do x <- c
   y <- f x
   g y             =  do y <- (do x <- c
                                  f x)
                         g y                             (ML3)
```

Chapter 11.4

Predefined Monads

Predefined Monads in Haskell

We consider a selection of predefined monads:

- Identity monad
- List monad
- Maybe monad
- Map monad
- State monad
- Input/Output monad

...but there are many more of them predefined in Haskell:

- Writer monad
- Reader monad
- Failure monad
- ...

As a Rule of Thumb

...when making a 1-ary type constructor a monad, then:

- `(>>=)` will be defined to unpack the value of the first argument, map the second argument over it, and return the packed result this yields.
- `return` will be defined in the most straightforward way to lift the argument value to its monadic counterpart.
- `(>>)` and `fail` are usually not to be implemented afresh. Usually, their default implementations provided in `Monad` are just fine.

  If the default implementations of `(>>)` and `fail` are used, this means for

  - `(>>)`: the first argument is evaluated and dropped, the second argument is evaluated and returned as result.
  - `fail`: the computation stops by calling `error` with some appropriate error message.

Chapter 11.4.1

The Identity Monad

The Identity Monad

...the identity monad (conceptually the simplest monad):

```
newtype Id a = Id a

instance Monad Id where
  (Id x) >>= f   =  f x
  return         =  Id
```

Note:

- `Id`: 1-ary type constructor, i.e., `Id a` denotes a type.
- `Id`: 1-ary data (or value) constructor, i.e., `Id v`, `v :: a`, denotes a value: `Id v :: Id a`.
- `(>>)` and `fail` are implicitly defined by their default implementations.
- ```
  (>>=)  :: Id a -> (a -> Id b) -> Id b
  return :: a -> Id a
  (>>)   :: Id a -> Id b -> Id b
  ```
  ```

Notes on the Identity Monad (1)

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
```

The instance declaration for `Id` with added type information:

```
instance Monad Id where
  Id x  >>=     f      =   f x  -- yields an (Id b)-value
:: Id a   :: a -> Id b   :: Id b
  return x             =   Id x -- yields an (Id a)-value
        :: a               :: Id a
```

Recall the overloading of `Id` (`newtype Id a = Id a`):

- `Id` followed by `x`: `Id` is data (or value) constructor.
- `Id` followed by `a` or `b`: `Id` is type constructor.

Notes on the Identity Monad (2)

Intuitively

- The identity monad maps a type to itself.
- It represents the trivial state, in which no actions are performed, and values are returned immediately.
- It is useful because it allows to specify computation sequences on values of its type (cf. Chapter 11.5.1)
- The operation `(>@>)` becomes for the identity monad forward composition of functions `(>.>)` (= `(>>;)`):

```
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
g >.> f = f . g
```

- Forward composition of functions `(>.>)` is associative with unit element `id`.

Lemma 11.4.1.1 (Monad Laws)

Instance `Id` of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

Chapter 11.4.2

The List Monad

The List Monad

...the list monad:

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
  fail s   = []
```

Note:

- `concat` and `map` are from the Standard Prelude.
- `[]`: 1-ary type constructor, i.e., `[a]` denotes a type.
- `[]`: 1-ary data (or value) constructor, i.e., `[x]`, `x :: a`, denotes a value: `[x] :: [a]`; in particular, `[]` denotes a value, the empty list.
- `(>>)` is implicitly defined by its default implementation; the default implementation of `fail` is overwritten.
- ```
  (>>=)  :: [] a -> (a -> [] b) -> [] b
  return :: a -> [] a
  (>>)   :: [] a -> [] b -> [] b
  ```
  ```

Monad Laws for []

Lemma 11.4.2.1 (Monad Laws)

Instance [] of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

For convenience, we recall from the Standard Prelude:

```
concat      :: [[a]] -> [a]
concat lss = foldr (++) [] lss

concat [[1,2,3],[4],[5,6]] ->> [1,2,3,4,5,6]
```

Notes on the List Monad

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for [] with added type information:

$$\texttt{instance Monad [] where}$$

$$\underbrace{\texttt{xs}}_{::\,[]\ \texttt{a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{::\,\texttt{a}\,\texttt{->}\,[]\ \texttt{b}} = \underbrace{\texttt{concat}\ \underbrace{\texttt{(map f xs)}}_{::\,[]\,([]\ \texttt{b})}}_{::\,[]\ \texttt{b}} \qquad \texttt{-- yields a [b]-list}$$

$$\texttt{return}\ \underbrace{\texttt{x}}_{::\,\texttt{a}} = \underbrace{\texttt{[x]}}_{::\,[]\ \texttt{a}} \qquad \texttt{-- yields the singleton list [x]}$$

$$\texttt{fail}\ \underbrace{\texttt{s}}_{::\,\texttt{String}} = \underbrace{\texttt{[]}}_{::\,[]\ \texttt{a}} \qquad \texttt{-- yields the empty list []}$$

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
Part IV
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
11.1
11.2
11.3
11.4

Using the Type Constructor [] as a Monad (1)

Examples:

```
ls = [1,2,3] :: [] Int
f = \n -> [(n,odd(n))] :: Int -> [] (Int,Bool)
g = \n -> [x*n | x <- [1.5,2.5,3.5]] :: Int -> [] Float
h = \n -> [1..n] :: Int -> [] Int

h 3 >>= f
  ->> ls >>= f
  ->> concat [ [(1,True)], [(2,False)], [(3,True)] ]
  ->> [(1,True),(2,False),(3,True)] :: [] (Int,Bool)

h 3 >>= g
  ->> ls >>= g
  ->> concat [ [ x*n | x <- [1.5,2.5,3.5] ] | n <- [1,2,3] ]
  ->> concat [ [1.5*1,2.5*1,3.5*1], [1.5*2,2.5*2,3.5*2],
               [1.5*3,2.5*3,3.5*3] ]
  ->> concat [ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] ]
  ->> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] Float
```

Using the Type Constructor [] as a Monad (2)

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for [] with added type information:

$$\texttt{instance Monad [] where}$$

$$\underbrace{\texttt{xs}}_{\texttt{:: [] a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{\texttt{:: a -> [] b}}\ = \underbrace{\texttt{concat}\ \underbrace{\texttt{(map f xs)}}_{\texttt{:: [] ([] b)}}}_{\texttt{:: [] b}} \quad \texttt{-- yields a [b]-list}$$

$$\texttt{return}\ \underbrace{\texttt{x}}_{\texttt{:: a}}\ = \underbrace{\texttt{[x]}}_{\texttt{:: [] a}} \quad \texttt{-- yields the singleton list [x]}$$

$$\texttt{fail}\ \underbrace{\texttt{s}}_{\texttt{:: String}}\ = \underbrace{\texttt{[]}}_{\texttt{:: [] a}} \quad \texttt{-- yields the empty list []}$$

Examples:

```
ls = [1,2,3] :: [] Int
f = \n -> [(n,odd(n))] :: Int -> [] (Int,Bool)
g = \n -> [x*n | x <- [1.5,2.5,3.5]] :: Int -> [] Float
h = \n -> [1..n] :: Int -> [] Int

h 3 >>= f ->> ls >>= f ->> concat [ [(1,True)], [(2,False)], [(3,True)] ]
          ->> [(1,True),(2,False),(3,True)] :: [] (Int,Bool)

h 3 >>= g ->> ls >>= g ->> concat [ [ x*n | x <- [1.5,2.5,3.5] ] | n <- [1,2,3] ]
          ->> concat [ [1.5*1,2.5*1,3.5*1], [1.5*2,2.5*2,3.5*2], [1.5*3,2.5*3,3.5*3] ]
          ->> concat [ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] ]
          ->> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] Float
```

The List Monad Reconsidered

...the list monad can equivalently be defined by:

```
instance Monad [] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  [] >>= f = []
  return x = [x]
  fail s   = []
```

Note: For the list monad the operations `(>>=)` and `return` have the types:

```
(>>=)  :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

List Monad and List Comprehension

...the list monad and list comprehension are closely related:

```
do x <- [1,2,3]
   y <- [4,5,6]
   return (x,y)
->> [(1,4),(1,5),(1,6),
     (2,4),(2,5),(2,6),
     (3,4),(3,5),(3,6)]
```

In fact, the following expressions are equivalent:

Proposition 11.4.2.2

```
[(x,y) | x <- [1,2,3], y <- [4,5,6] ] <=>
                                  do x <- [1,2,3]
                                     y <- [4,5,6]
                                     return (x,y)
```

...list comprehension is syntactic sugar for monadic syntax!

List comprehension

...as syntactic sugar for monadic syntax.

We have:

Lemma 11.4.2.3

```
[f x | x <- xs] <=> do x <- xs; return (f x)
```

Lemma 11.4.2.4

```
[a | a <- as, p a] <=>
  do a <- as; if (p a) then return a else fail ""
```

Homework

Prove by stepwise evaluation the equivalences stated in:

1. Proposition 11.4.2.2
2. Lemma 11.4.2.3
3. Lemma 11.4.2.4

Chapter 11.4.3

The Maybe Monad

The Maybe Monad

...the Maybe monad:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return         = Just
  fail s         = Nothing
```

Note:

- `(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b`
  `return :: a -> Maybe a`
  `(>>)   :: Maybe a -> Maybe b -> Maybe b`
- The Maybe monad is useful for computation sequences that can produce a result, but might also produce an error.

Monad Laws for Maybe

Lemma 11.4.3.1 (Monad Laws)

Instance Maybe of Monad satisfies the three monad laws ML1, ML2, and ML3.

Recall that Maybe is also a predefined instance of Functor:

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Lemma 11.4.3.2 (Monad/Functor Laws)

Instance Maybe of Monad and Functor satisfies law MFL (of Chap. 11.2).

Notes on the Maybe Monad

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for `Maybe` with added type information:

`instance Monad Maybe where`

$$\underbrace{\texttt{Just x}}_{::\,\texttt{Maybe a}}\ \texttt{>>=}\ \underbrace{\texttt{k}}_{::\,\texttt{a -> Maybe b}}\ =\ \underbrace{\texttt{k x}}_{::\,\texttt{Maybe b}}\quad \texttt{-- yields a Just-value}$$

$$\underbrace{\texttt{Nothing}}_{::\,\texttt{Maybe a}}\ \texttt{>>=}\ \underbrace{\texttt{k}}_{::\,\texttt{a -> Maybe b}}\ =\ \underbrace{\texttt{Nothing}}_{::\,\texttt{Maybe b}}\quad \texttt{-- yields the Nothing-value}$$

$$\texttt{return}\ \underbrace{\texttt{x}}_{::\,\texttt{a}}\ =\ \underbrace{\texttt{Just x}}_{::\,\texttt{Maybe a}}\quad \texttt{-- yields the Just-value}$$

$$\texttt{fail}\ \underbrace{\texttt{s}}_{::\,\texttt{String}}\ =\ \underbrace{\texttt{Nothing}}_{::\,\texttt{Maybe a}}\quad \texttt{-- yields the empty list}$$

Using the Maybe Monad: Error Handling (1)

...or: How to compose functions with monadic value ranges.

Let f′ and g′ be two functions of type:

```
f' :: a -> b
g' :: b -> c
```

Obviously, composing f′ and g′ sequentially is straightforward:

```
h' :: a -> c
h' = (g' . f')

h' x ->> (g' . f') x ->>  g' (f' x)
```

Using the Maybe Monad: Error Handling (2)

If the computations of f' and g' can fail, this can be taken care of by replacing f' and g' by two new functions f and g embedding the computation into the Maybe type:

```
f :: a -> Maybe b                    -- f replaces f'
g :: b -> Maybe c                    -- g replaces g'
```

Unlike f' and g', however, f and g can not straightforwardly be sequentially composed:

```
h :: a -> Maybe c                  -- "h = (g . f)":
h x = case (f x) of              -- Composing f and g
       Nothing -> Nothing          -- requires nested
       Just y  -> case (g y) of    -- case clauses
                   Nothing -> Nothing
                   Just z  -> Just z
```

Though possible, the explicit nesting of cases to sequentially compose f and g is inconvenient and tedious.

Using the Maybe Monad: Error Handling (3)

Step 1: Hiding nestings.

...embedding f′ and g′ into the Maybe type gets a lot easier by exploiting the monad property of Maybe: Using the monadic sequencing operations for composing f and g allows:

```
h :: a -> Maybe c                    -- "h = (g . f)"
h x = f x >>= \y -> g y >>= \z -> return z
```

or, equivalently, using the do notation:

```
h :: a -> Maybe c                    -- "h = (g . f)"
h x = do y <- f x
         z <- g y
         return z
```

...the 'nasty' error checks are now hidden in the implementation of the bind operation (>>=) of the Maybe monad.

Using the Maybe Monad: Error Handling (4)

Step 2: Hiding the bind operation (>>=).

Note that the sequence of monad operations:

```
f x >>= \y -> g y >>= \z -> return z
```

can be simplified to:

```
f x >>= \y -> g y >>= \z -> return z
    <=> (simplification by currying)
f x >>= \y -> g y >>= return
   <=> (monad law for return)
f x >>= \y -> g y
   <=> (simplification by currying)
f x >>= g
```

Hence, `h x` (“= `g (f x)`”) is equivalent to `f x >>= g`.

Using the Maybe Monad: Error Handling (5)

...making use of this observation and introducing function:

```
composeM :: Monad m => (b -> m c) ->
                           (a -> m b) -> (a -> m c)
(g `composeM` f) x = f x >>= g
```

allows an even more pleasing notation for composing f and g:

```
h :: a -> Maybe c                -- "h = (g . f)"
h = (g `composeM` f)
```

Hence, we obtain:

```
(g composeM f)
```

as the monadic notational counterpart of sequentially composing $\mathtt{f}'$ and $\mathtt{g}'$:

$(\mathtt{g}' \ . \ \mathtt{f}')$

Using the `Maybe` Monad: Error Handling (6)

Overall: Using monadic sequencing

`f x >>= g` (or equivalently: `(g `composeM` f) x`)

for embedding the composition of f' and g' into the `Maybe` type preserves the original syntactical form of composing f' and g':

`(g' . f') x = g' (f' x)`

in almost a 1-to-1 kind:

`(g composeM f) x = f x >>= g`

Chapter 11.4.4

The Either Monad

Homework

1. Make type constructor `(Either a)` an instance of `Monad`.
2. Provide (most general) type information for the defining equations of the monad operations `(>>=)`, `(>>)`, `return`, and `fail` of `(Either a)`.
3. Prove that `(Either a)` satisfies the monad laws.

Chapter 11.4.5

The Map Monad

The Map Monad

...the map monad:

```
instance Monad ((->) d) where
  h >>= f  = \x -> f (h x) x
  return x = \_ -> x
```

Note:

```
(>>=)  :: ((->) d) a -> (a -> ((->) d) b) -> ((->) d) b
return :: a -> ((->) d) a
(>>)   :: ((->) d) a -> ((->) d) b -> ((->) d) b
```

Lemma 11.4.5.1 (Monad Laws)

Instance `((->) d)` of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

Example (w/ String for d, Int for a, (Bool,String) for b) (1)

```
(>>=)  :: ((->) d) a -> (a -> ((->) d) b) -> ((->) d) b
  ( ≙ (>>=) :: (d -> a) -> (a -> (d -> b)) -> (d -> b) )
h >>= f  = \x -> f (h x) x

h_length :: ((->) String) Int
  ( ≙ h_length :: String -> Int )
h_length = length
f_cp_p :: Int -> ((->) String) ((,) Bool String)
  ( ≙ f_cp_p :: Int -> (String -> (Bool,String) )
f_cp_p n s = (,) (mod n 2 == 1) (copy n s)
 where copy n s = if n > 0 then s ++ " " ++ copy (n-1) s else ""
g :: ((->) String) ((,) Bool String)
  ( ≙ g :: String -> (Bool,String) )
g = \s -> f_cp_p (h_length s) s
 ( ≙ g s = (mod (length s) 2 == 1,copy (length s) s) )

h_length >>= f_cp_p
 ->> (\x -> f_cp_p (h_length x) x)    ( = g )

(h_length >>= f_cp_p) "Fun"
 ->> ... ->> (True,"Fun Fun Fun")
```

Example (w/ String for d, Int for a, (Bool,String) for b) (2)

...in more detail:

```
h_length >>= f_cp_p
 ->> (\x -> f_cp_p (h_length x) x)
  = g    ( :: String -> (Bool,String) )

(h_length >>= f_cp_p) "Fun"
 ->> (\x -> f_cp_p (h_length x) x) "Fun"
  = g "Fun"
 ->> (mod (length "Fun") 2 == 1,copy (length "Fun") "Fun")
 ->> (mod 3 2 == 1,copy 3 "Fun")
 ->> (True,"Fun Fun Fun")      ( :: (Bool,String) )
```

Example (w/ String for d, Int for a, (Bool,String) for b) (3)

```
(>>=)  :: ((->) d) a -> (a -> ((->) d) b) -> ((->) d) b
h >>= f  = \x -> f (h x) x

return :: a -> ((->) d) a ( ≙ return :: Int -> ((->) String) Int)
return x = \_ -> x          ≙ return :: Int -> (String -> Int) )

return 0 = \_ -> 0    ( :: String -> Int )

return 0 >>= f_cp_p
 ->> \x -> f_cp_p ((return 0) x ) x
 ->> \x -> f_cp_p (\_ -> 0) x) x ( :: String -> (Bool,String) )

(return 0 >>= f_cp_p) "Fun"
 ->> (\x -> f_cp_p ((return 0) x ) x) "Fun"
 ->> f_cp_p ((return 0) "Fun" ) "Fun"
 ->> f_cp_p ((\_ -> 0) "Fun") "Fun"
 ->> f_cp_p 0 "Fun"
 ->> (mod 0 2 == 1,copy 0 "Fun")
 ->> (False,"")        ( :: (Bool,String) )

(return 1 >>= f_cp_p) "Fun" ->> ... ->> (True,"Fun")
(return 2 >>= f_cp_p) "Fun" ->> ... ->> (False,"Fun Fun")
(return 3 >>= f_cp_p) "Fun" ->> ... ->> (True,"Fun Fun Fun")
```

Example (w/ String for d, Int for a) (4)

```
(>>=)  :: ((->) d) a -> (a -> ((->) d) b) -> ((->) d) b
h >>= f  = \x -> f (h x) x

return :: a -> ((->) d) a ( ≙ return :: Int -> ((->) String) Int)
return x = \_ -> x          ≙ return :: Int -> (String -> Int) )

return 3 = \_ -> 3    ( :: String -> Int )

h_length >>= return
 ->> \x -> return (h_length x) x
 ->> \x -> return (length x) x
 ->> \x -> (\_ -> length x) x    ( :: String -> Int )

(h_length >>= return) "Fun"
 ->> (\x -> (return (h_length x) x)) "Fun"
 ->> return (h_length "Fun") "Fun"
 ->> return (length "Fun") "Fun"
 ->> return 3 "Fun"
 ->> (\_ -> 3) "Fun"
 ->> 3    ( :: Int )
```

Homework

1. Recall the monad operations:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
```

   Add (most general) type information for the instance declaration of `((->) d)`:

```
instance Monad ((->) d) where
  h  >>=  f     =    \x -> f (h x) x
  return x      =    \_ -> x
```

2. Evaluate stepwise:
   2.1 `(return 2 >>= f_cp_p) "Fun"`
   2.2 `(h_length >>= return) "Fun Prog"`
   2.3 `(h_length >>= return >>= f_cp_p) "Fun"`

Chapter 11.4.6

The State Monad

Objective

...modelling global state and side effects by means of functions, which,

- applied to some initial state s yield a new state s′ as part of the overall result of the computation.

The State Monad

...the state monad:

```
newtype State st a = St (st -> (st,a))

instance Monad (State st) where
  (St h) >>= f = St (\s -> let (s',x) = h s
                    :: st        St f'  = f x
                             in f' s')
                                :: (st,b)
    -- Applying map h :: (st -> (st,a)) to state s :: st
    -- yields a pair (s',x) :: (st,a) onto whose 2nd compo-
    -- nent x :: a map f :: a -> (State st) b is applied.
    -- This yields a state value St f' :: (State st) b,
    -- whose map value f' :: st -> (st,b) is applied to
    -- s' :: st yielding a pair f' s' :: (st,b) as required.

  return x = St (\s -> (s,x))
        :: a    :: st :: (st,a)
    -- x :: a and every state s :: st are identically mapped.
```

Monad Laws for (State st)

Note: For the state monad (State st) the monad operations (>>=) and return have the types:

```
(>>=)  :: (State st) a -> (a -> (State st) b) -> (State st) b
return :: a -> (State st) a
(>>=)  :: (State st) a -> (State st) b -> (State st) b
```

Lemma 11.4.6.1 (Monad Laws)

Instance (State st) of Monad satisfies the three monad laws ML1, ML2, and ML3.

Notes on the State Monad

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
c >>= k = ... :: m b
return :: (Monad m) => a -> m a
return x = ... :: m a
```

The instance declaration for `(State st)` with added type information:

```
instance Monad (State st) where
      St h       >>=         f
 :: (State st) a    :: a -> (State st) b
             = St (\s -> let ... in f' s')   -- constructing
                   :: st      :: (st,b)      -- a proper state
                      :: st -> (st,b)        -- value using m
                    :: (State st) b          -- and f.
  return x   =  St (\s -> (s,x))     -- constructing a proper
        :: a     :: (State st) a     -- state value using x
                                     -- in the simplest way.
```

Intuitively

...state transformers

- are mappings `m` of some type `(st -> (st,a))`,
  i.e., `m :: st -> (st,a)`.
- map (or transform) global (internal program) states of type `st` into (possibly modified) new states of type `st` while additionally computing a result of type `a`.
- map an argument state `s` of some type `st` to a pair of a (possibly modified) result state s' of type `st` and a value `v` of some type `a`,
  i.e., `m s =>> (s',v)`, `s :: st`, `s' :: st`, `v :: a`.

The State Monad

...specialized for some concrete state (component) type.

Let CStT (reminding to 'Concrete State Type') be some concrete type (e.g., Int, [String],...):

```
newtype State' a = St' (CStT -> (CStT,a))

instance Monad State' where
  St' m >>= f = St' (\cs -> let (cs',x) = m cs
                   :: CStT       St' f' = f x
                              in f' cs')
                              :: (CStT,b)

  return x     = St' (\cs -> (cs,x))
       :: a          :: CStT  :: (CStT,a)
```

Note: State' is a 1-ary type constructor while State is a 2-ary type constructor.

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
Part IV
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
11.1
11.2
11.3
11.4

Monad Laws for (State′)

Note: For the state monad State′ the monad operations (>>=) and return have the types:

```
(>>=)  :: State' a -> (a -> State' b) -> State' b
return :: a -> State' a
```

Lemma 11.4.6.2 (Monad Laws)

Instance State′ of Monad satisfies the three monad laws ML1, ML2, and ML3.

The State Monad Reconsidered (1)

...sometimes renaming objects helps getting things clear(er).

Think about `st_otw` as a type variable where the values of appropriate type instances of `st_otw` describe or model the

- State of the World (St_otW).

The sequencing operation `(>>=)` of the state monad `(State st_otw)` allows then to transform a current state of the world into a new state of the world, i.e., to

- transform (the description of) the state of the world it is currently in into (the description of) the world it is in after the transformation, i.e., (the description of) the new state the world is in afterwards.

Intuitively, this suggests for a state transformer:

```
state_transformer :: st_otw -> st_otw
```

Note: `(State st_otw)`, `(>>=)` make this a bit more complex.

The State Monad Reconsidered (2)

```
newtype (State stotw) a = St (stotw -> (stotw,a))
instance Monad (State stotw) where
 St h >>= f
  = St (\current_state ->
         let (intermediate_state,x) = h current_state
             St g = f x
             (new_state,z) = g intermediate_state
         in (new_state,z)
return x = St (\current_state -> (current_state,x))
```

Note resp. compare:

- `(>>=) :: (State stotw) a -> (a -> (State stotw) b) -> (State stotw) b`

  `return :: a -> (State stotw) a`
- ```
  (g . f) = (f; g) = \x -> let intermediate = f x
                               y = g intermediate
                           in y          -- y = g (f x)
  ```
  ```

Chapter 11.4.7

The Input/Output Monad

The Input/Output Monad

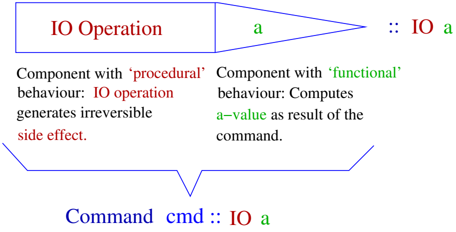
```
instance Monad IO where      (Impl. intern. hidden)
 (>>=)  :: IO a -> (a -> IO b) -> IO b
 return :: a -> IO a
 (>>)   :: IO a -> IO b -> IO b
 fail   :: String -> IO a
```

Note:

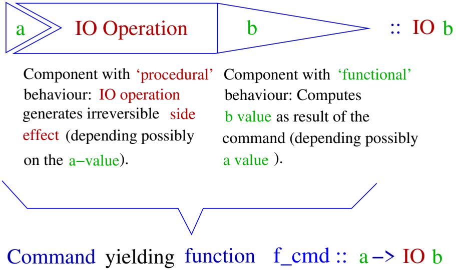
- `IO`-values are so-called IO-commands (or commands).
- Commands have a procedural effect (i.e., reading or writing) and a functional effect (i.e., computing a value).
- `(>>=)`: If `p`, `q` are commands, then `p >>= q` is a composed command that first executes `p`, thereby performing a read or write operation and yielding an `a`-value `x` as result; subsequently `q` is applied to `x`, thereby performing a read or write operation and yielding a `b`-value `y` as result.
- `return`: Lifts an `a`-value to an `IO a`-value w/out performing any input or output operation.

Illustrating the Nature of Commands

Command `cmd :: IO a`

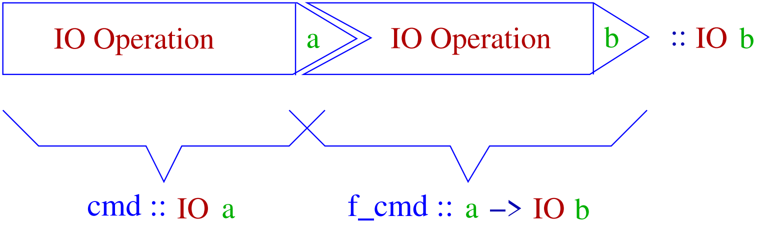


Command yielding function `f_cmd :: a -> IO b`



Illustrating

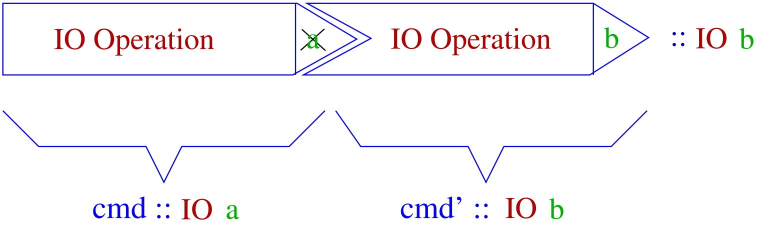
...the operational meaning of `(cmd >>= f_cmd)`:



`cmd >>= f_cmd` $\widehat{=}$ `cmd >>= \x -> f_cmd x`

Illustrating

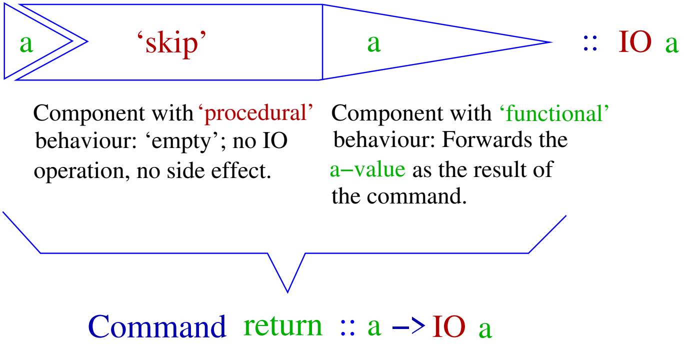
...the operational meaning of `(cmd >> cmd')`:



$$\texttt{cmd >> cmd}' \;\widehat{=}\; \texttt{cmd >> \textbackslash\_ -> cmd}'$$

Illustrating

...the operational meaning of `return`:



The Type

...of all read commands is

- `(IO a)` (for type instances `a` whose values can be read).

  The `a`-value into which the read value is transformed serves as the (formally required and actually wanted) result of read operations.

...of all write commands is

- `(IO ())`, where `()` is the singleton null tuple type with the single unique element `()`.

  `()` as (the one and only) value of the null tuple type `()` serves as the formally required result of write operations.

Lemma 11.4.7.1 (Monad Laws)

Instance `IO` of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

Input/Output and State Monad

...the input/output monad is similar in spirit to the state monad: It passes around the "state of the world!"

For a suitable type `World` whose values represent the

- states of the world

interactive programs (or IO-programs) can informally be considered functions of a type `IO` with:

- "`type IO = (World -> World)`"

In order to reflect that interactive programs do not only modify the state of the world but may also return a result, e.g., the `Int`-value of a sequence of characters that has been read from the keyboard and interpreted as an integer, this leads to changing the informal type of IO-programs from `IO` to `(IO a)`:

- "`type IO a = (World -> (World,a))`"

The Input/Output Monad (1)

...allows switching from a batch-like handling of input/output:

Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, p. 245.

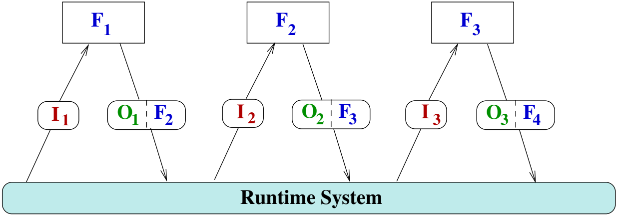
where

- all input data must be provided at the very beginning
- there is no interaction between a program and a user (i.e., once called there is no opportunity for the user to react on a program's response and behaviour)

by a...

The Input/Output Monad (2)

...truly interactive handling of input/output in terms of sequentially composed dialogue components, while preserving referential transparency as far as possible:



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, p. 253.

Note that input/output operations are a major source for side effects: read statements e.g. will yield different values for every call which directly causes the loss of referential transparency.

Examples: Simple IO Programs (1)

...a question/response interaction with a user:

```
ask :: String -> IO String
ask question = do putStrLn question
                  getLine

interAct :: IO ()
interAct =
    do name <- ask "May I ask your name?"
       putStrLine ("Welcome " ++ name ++ "!")
```

Examples: Simple IO Programs (2)

...input/output from and to files:

```
type FilePath = String  -- file names according
                        -- to the conventions of
                        -- the operating system

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath -> IO String
isEOF      :: FilePath -> IO Bool

interAct :: IO ()
interAct = do putStr "Please input a file name: "
              fname <- getLine
              contents <- readFile fname
              putStr contents
```

Examples: Simple IO Programs (3)

...note the relationship of do-notation

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
```

and (canonic) monadic operations:

```
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```

Note also sometimes (subtle) difference in result types:

```
Main>putStr ('a':('b':('c':[])))  Main>putChar (head ['x','y','z'])
             ->> abc :: IO ()               ->> x :: IO ()
```

but

```
Main>('a':('b':('c':[])))         Main>head ['x','y','z']
             ->> "abc" :: [Char]            ->> 'x' :: Char

Main>print "abc"                    Main>print 'x'
             ->> "abc" :: IO ()             ->> 'a' :: IO ()
```

Examples: Simple IO Programs (4)

...the sequence of output commands

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
```

is equivalent to:

```
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```

Examples: Simple IO Programs (5)

...the sequence of input/output commands with local declarations within a do-construct

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   let rev1 = reverse line1
                   let rev2 = reverse line2
                   putStrLn rev2
                   putStrLn rev1
```

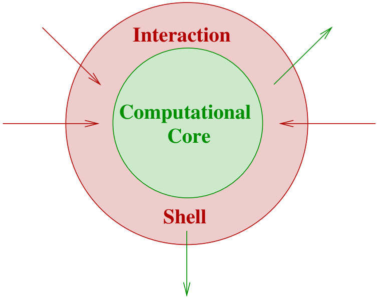
is equivalent to the following one without:

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   putStrLn (reverse line2)
                   putStrLn (reverse line1)
```

In Closing (1)

...monadic input/output in Haskell allows us to conceptually think of a Haskell program as consisting of a

- a purely functional computational core and
- a procedural-like interaction shell.



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell.* Pearson, 2004, p. 89.

In Closing (2)

...the monad concept of Haskell allows to

- conceptually separate functions belonging to the
  - computational core (pure functions)
  - interaction shell (impure functions, i.e., performing input/output operations causing side effects).

  by assigning different types to them:

  $\leadsto$ `Int`, `Real`, `String`,... vs. `IO` Int, `IO` Real, `IO` String,...
  with type constructor `IO` a pre-defined monad.

- precisely specify the evaluation order of functions of the interaction shell (i.e., of basic input/output primitives provided by Haskell) by using the monadic sequencing operations.

...see e.g. lecture notes of LVA 185.A03 Funktionale Programmierung for further details and examples.

Chapter 11.5

Monadic Programming

Outline

...we consider three examples for illustration:

1. Folding trees by adding the values of their numerical labels.
2. Numbering tree labels (and overwriting the original labels).
3. Renaming tree labels by the number of their occurrences.

Chapter 11.5.1

Folding Trees

The Setting (1)

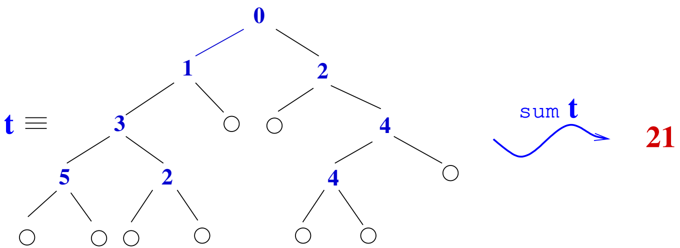
Given:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Objective:

- Write a function that computes the sum of the values of all labels of a tree of type `Tree Int`.

  Illustration:



The Setting (2)

Means:

Two functional approaches

- w/out monads
- w/ monads

respectively, for comparison.

1st Approach: Straightforward w/out Monads

...using a recursive function:

```
sum :: Tree Int -> Int
sum Nil              = 0
sum (Node n t1 t2) = n + sum t1 + sum t2
```

Note:

- The evaluation order of the right-hand term of the (non-trivial) defining equation of `sTree` is not fixed; only data dependencies need to be respected.
- This leaves interpreter and compiler a degree of freedom in picking an evaluation order.
- This freedom can not be broken by a programmer by using a specific right-hand side term:

```
sum (Node n t1 t2) = n + sum t1 + sum t2
sum (Node n t1 t2) = sum t2 + n + sum t1
...
sum (Node n t1 t2) = sum t2 + sum t1 + n
```

2nd Approach: Using the Identity Monad

...using the identity monad Id:

```
sum' :: Tree Int -> Id Int
sum' Nil = return 0
sum' (Node n t1 t2) =
 do s2  <- sum' t2          -- Evaluating right subtree
    num <- return n         -- Bounding n :: Int to num
    s1  <- sum' t1          -- Evaluating left subtree
    return (s2+num+s1)      -- Yielding Id (num+s1+s2) ::
                            -- Id Int as result
```

Note:

- The evaluation order of the defining 'equations' for s2, n, and s1 is explicitly fixed; there is no degree of freedom for the sequence in which values are bound to them.
- Changing their order allows the programmer to enforce a different evaluation order.
- Note, this does not apply to evaluating s2+num+s1.

The Identity Monad

Recall the identity monad `Id`:

```
newtype Id a = Id a

instance Monad Id where
  (Id x) >>= f = f x
  return       = Id
```

Note:

- `Id`: 1-ary type constructor, i.e., `Id a` denotes a type.
- `Id`: 1-ary data (or value) constructor, i.e., `Id v`, `v :: a`, denotes a value: `Id v :: Id a`.

Illustrating the Imperative Flavour of sum′

...unlike sum, sum′ enjoys an 'imperative' flavour quite similar to sequentially sequencing assignment statements of some imperative programming language:

Imperative

```
s2  := sumTree t2;
s1  := sumTree t1;
num := n;
return (s2+s1+num);
```

Monadic

```
do s2  <- sumTree t2
   s1  <- sumTree t1
   num <- return n
   return (s2+s1+num)
```

3rd Approach: Using extract and Monad Id

...using an extraction function to allow a function sum″ of type (Tree Int -> Int):

```
extract :: Id a -> a
extract (Id x) = x
```

This enables:

```
sum″ :: Tree Int -> Int
sum″ = extract . sum′
```

Example:

```
t = (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil))
sum″ t ->> (extract . sum′) t
       ->> extract (sum′ t)
       ->> extract (Id 15)
       ->> 15
```

Chapter 11.5.2

Numbering Tree Labels

The Setting

Given:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Objective:

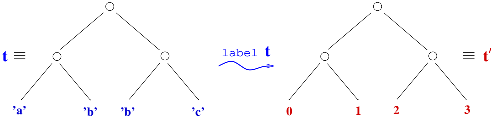
- Replace the labels of leafs by continuous natural numbers.

  Illustration: The tree value `t :: Tree Char`:

  ```
  t = Branch (Branch (Leaf 'a') (Leaf 'b'))
             (Branch (Leaf 'b') (Leaf 'c'))
  ```

  shall be transformed into the tree value `t' :: Tree Int`:

  ```
  t' = Branch (Branch (Leaf 0) (Leaf 1))
              (Branch (Leaf 2) (Leaf 3))
  ```



The Setting (2)

Means:

Two functional approaches

- w/out monads
- w/ monads

respectively, for comparison.

1st Approach: Straightforward w/out Monads

...using a pair of functions, one of which a recursive supporting function:

```
label :: Tree a -> Tree Int
label t = snd (lab t 0)

lab :: Tree a -> Int -> (Int, Tree Int)
lab (Leaf a) n = (n+1, Leaf n)
lab (Branch t1 t2) n
    = let (n1,t1′) = lab t1 n
          (n2,t2′) = lab t2 n1
      in (n2, Branch t1′ t2′)
```

Note: The solution is simple and straightforward but passing the counter value `n` through the incarnations of `lab` is tedious and intricate.

2nd Approach: Using the State Monad (1)

...using the pattern of the state monad State′:

```
newtype Label a = Lab (Int -> (Int,a))

instance Monad Label where
 Lab lt >>= flt = Lab $ \n -> let (n',x)  = lt n
                                  Lab lt' = flt x
                              in lt' n'
 return x       = Lab (\n -> (n,x))
```

Note:

- The $-operator in the defining equation of (>>=) can be dropped by bracketing expr. `\n -> let ... in lt' n'`.
- For the state monad Label the monad operations (>>=) and return have the types:

```
(>>=)  :: Label a -> (a -> Label b) -> Label b
return :: a -> Label a
```

2nd Approach: Using the State Monad (2)

...the renaming of labels can now be achieved as follows:

```
label′ :: Tree a -> Tree Int
label′ t = let Lab lt = lab′ t
           in snd (lt 0)

lab′ :: Tree a -> Label (Tree Int)
lab′ (Leaf a) = do n <- get_label
                   return (Leaf n)
lab′ (Branch t1 t2) = do t1′ <- lab′ t1
                         t2′ <- lab′ t2
                         return (Branch t1′ t2′)

get_label :: Label Int
get_label = Lab (\n -> (n+1,n))
```

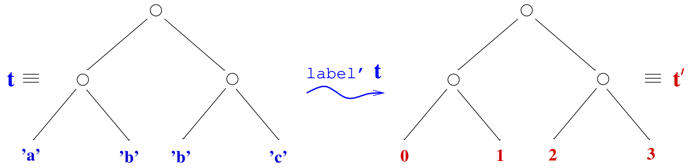
2nd Approach: Using the State Monad (3)

Example: Applying `label′` to tree value `t`:

```
t = Branch (Branch (Leaf ‘a‘) (Leaf ‘b‘))
           (Branch (Leaf ‘b‘) (Leaf ‘c‘))
```

we get as desired:

```
label′ t ->> Branch (Branch (Leaf 0) (Leaf 1))
                    (Branch (Leaf 2) (Leaf 3))
         ≡  t′
```



Homework

Provide (most general) type information for the defining equations of

1. the operations
   1.1 `(>>=)`
   1.2 `return`

   of the state instance declaration of `Label`.
2. the functions
   2.1 `label′`
   2.2 `lab′`
   2.3 `get_label`

   of the monadic solution of the numbering problem.

Chapter 11.5.3

Renaming Tree Labels

The Setting

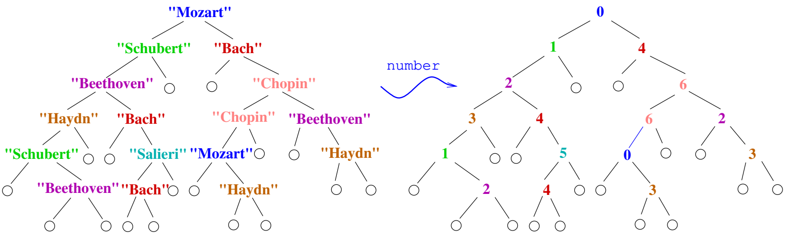
Given:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Objective:

- Rename labels of equal a-value by the same natural number.

  Illustration:



Ultimate Goal

...a function `number` of type

```
number :: Eq a => Tree a -> Tree Int
```

solving this task using the state monad `State`.

Towards a Monadic Approach (1)

We start defining:

```
number_tree :: Eq a => Tree a -> State a (Tree Int)
number_tree Nil = return Nil
number_tree (Node x t1 t2) =
                 = do num <- number_node x
                      nt1 <- number_tree t1
                      nt2 <- number_tree t2
                      return (Node num nt1 nt2)
```

...post-poning the implementation of number_node.

Towards a Monadic Approach (2)

Additionally, we introduce a table type

```
type Table a = [a]
```

for storing pairs of the form

(<string>,<number of occurrences>)

In particular, the list (or table) value

```
[True,False]
```

encodes that `True` represents (or is associated with) `0` and `False` with `1`.

Mon. Approach: Using the State Monad (1)

...using the pattern of the state monad `State st`:

```
newtype State a b = St (Table a -> (Table a, b))

instance Monad (State a) where
  (St st) >>= f
     = St (\tab -> let (tab',y)     = st tab
                       (St transf) = f y
                   in transf tab')
   return x = St (\tab -> (tab, x))
```

Intuitively:

- Computing b-values: The (functional) result
- Updating tables: The side effect

...of the monadic operations.

Mon. Approach: Using the State Monad (2)

...providing the post-poned implementation of number_node:

```
number_node :: Eq a => a -> (State a) Int
number_node x = St (num_node x)

num_node :: Eq a => a -> (Table a -> (Table a, Int))
num_node x table
  | elem x table = (table,        lookup x table)
  | otherwise    = (table ++ [x], length table)
  -- num_node yields the position of x in the table:
  -- if x is stored in the table, using lookup; if
  -- not, after adding x to the table using length.

lookup :: Eq a => a -> Table a -> Int
lookup x table = ... -- Homework: Completing the im-
                     -- plementation of lookup.
```

Mon. Approach: Using the State Monad (3)

Putting the pieces together, number_tree is fully defined:

```
number_tree :: Eq a => Tree a -> State a (Tree Int)
number_tree Nil = return Nil
number_tree (Node x t1 t2)
                = do num <- number_node x
                     nt1 <- number_tree t1
                     nt2 <- number_tree t2
                     return (Node num nt1 nt2)
```

Note, for every value `t :: Eq a => Tree a`, e.g., the tree of the illustrating example, we can conclude (functional and hence) type correctness:

```
number_tree t :: State a (Tree Int)
              ≡ (State a) (Tree Int)
              ≡ ((State a) (Tree Int))
```

Mon. Approach: Using the State Monad (4)

...introducing an extraction function:

```
extract :: State a b -> b
extract (St st) = snd (st [])
```

we get the implementation of the initially envisioned function `number`:

```
number :: Eq a => Tree a -> Tree Int
number = extract . number_tree
```

Homework

Provide (most general) type information for the defining equations of

1. the operations
   1.1 `(>>=)`
   1.2 `return`

   of the state instance declaration of `(State a)`.
2. the functions
   2.1 `number`
   2.2 `number_tree`
   2.3 `number_node`
   2.4 `num_node`
   2.5 `lookup`

   of the monadic solution of the renaming problem.

Chapter 11.6

MonadPlus

Chapter 11.6.1

The Type Constructor Class MonadPlus

The Type Constructor Class MonadPlus

...monads with an appropiate 'zero' element and 'plus' operation can be instances of the type constructor class MonadPlus.

Type Constructor Class MonadPlus

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

MonadPlus Laws

```
m >>= (\x -> mzero) = mzero          (MPL1)
mzero >>= m         = mzero          (MPL2)
m `mplus` mzero     = m              (MPL3)
mzero `mplus` m     = m              (MPL4)
```

Proper Instances of the Type Class MonadPlus

...must satisfy additionally to all monad laws the monadPlus laws, i.e., the laws for the 'zero' element and the 'plus' operation, which, intuitively, mean:

- `mzero` is left-zero and right-zero for `(>>=)`.
- `mzero` is left-unit and right-unit for `mplus`.

Proof obligation: As usual for type class instances, it is the programmer's obligation to prove that their instances of `MonadPlus` satisfy all monad and monadPlus laws.

Note: The `IO` monad can not be made an instance of `MonadPlus` because of the lack of an appropriate 'zero' element.

Chapter 11.6.2

The Maybe MonadPlus

The Maybe Instance of MonadPlus

...the Maybe instance of the type constructor class MonadPlus:

```
instance MonadPlus Maybe where
  mzero              = Nothing
  Nothing `mplus` ys = ys
  xs `mplus` ys      = xs
```

Lemma 11.6.2.1 (Maybe Instance of MonadPlus)

Instance Maybe of MonadPlus satisfies all monad and monadPlus laws.

Chapter 11.6.3

The List MonadPlus

The [] Instance of MonadPlus

...the list instance of the type constructor class MonadPlus:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Lemma 11.6.3.1 (List Instance of MonadPlus)

Instance [] of MonadPlus satisfies all monad and monadPlus laws.

Homework

1. Provide (most general) type information for
   1.1 the monadPlus laws MPL1, MPL2, MPL3, and MPL4.
   1.2 the defining equations of 'zero' element and 'plus' operation of the
      1.2.1 `Maybe` instance
      1.2.2 `[]` instance

      of `MonadPlus`.
2. Which of the other monads considered in Chapter 11.4 (Identity, Either, Map, State, Input/Output) could be reasonable instances of `MonadPlus`? Which of them are pre-defined instances?
   2.1 Provide instance declarations, where possible, together with (most general) type information for the defining equations of the MonadPlus operations.
   2.2 Prove that all instances satisfy the MonadPlus laws.

Chapter 11.7

Summary

Summary

Monads (i.e., instances of type constructor class `Monad`) combine features of

- functors and functional composition:
  `(>>=) :: m a -> (a -> m b) -> m b`
  c `>>=` k `>>=` k′ `>>=` k″ `>>=` ...

Monads are thus well-suited for

- structuring and sequencing evaluation steps

because they

- allow to specify sequential program parts systematically.
- offer an adequately high abstraction by decoupling the data type forming a monad (instance) from the structure of computation.
- support equational reasoning, e.g., by applying the monad laws.

On the Origins of the Notion Monad: Monads in Philosophy

...monad, derived from Greek *monas* meaning unit(y) (in German: Eins, Einheit).

Gottfried Wilhelm Leibniz (* 1646 in Leipzig; † 1716 in Hannover) used monad as a counterpart of

- 'atom' denoting like atom 'something indivisable'

to 'solve' (possibly more accurate: tackle) the so-called

- body-soul problem (in German: Leib-Seele-Problem)

evolving from the body-soul dualism in the the classical formulation of René Descartes (* 1596 in La Haye 50 km south of Tours, today Descartes; † 1650 in Stockholm).

Monads in Category Theory

Eugenio Moggi introduced/used the monad notion into

- category theory

and

- programming languages theory

as a means for describing the

- semantics of programming languages:

  Eugenio Moggi. Computational Lambda Calculus and Monads. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.

Monads in Functional Programming

...later on, the monad notion became particularly popular (w/out the background from philosophy and category theory) in the field of functional programming (Philip Wadler, 1992), especially because monads as in the sense of Haskell e.g.

- allow to introduce some useful aspects of imperative programming such as sequencing into functional programming,
- are well suited for smoothly integrating input/output into functional programming, as well as many other programming tasks and domains,
- provide a suitable interface between functional programming and programming paradigms with side effects, in particular, imperative and object-oriented programming,

...without breaking the functional paradigm!

On the other Hand

...the origin and connection of the monad notion to (often difficult considered) fields like

- philosophy, category theory, programming languages theory, programming languages semantics

might be responsible for awarding the monad notion an aura of something

- mysticly, wondrously that is difficult to grasp (*'once I will have understood monads, I will have understood functional programming'* letting monads appear the Holy Grail of functional programming).

Overall, this gives monads a mythical flavour.

On Constituting the Mythical Aura

Monads in Leibniz' Philosophy:

Definition (Gottfried Wilhelm Leibniz, 1714)

[Monadology, Paragraph 1]: The monad we want to talk about here is nothing else as a simple substance (German: Substanz), which is contained in the composite matter (German: Zusammengesetztes); simple means as much as: to be without parts.

Monads in Category Theory (cf. Saunders Mac Lane, 1971):

Definition (Eugenio Moggi, 1989)

[LICS'89]: A monad over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : Id_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ are natural transformations and the following equations hold:

$$\begin{array}{rcll} \mu_{TA}; \mu_A & = & T(\mu_a); \mu_A & \\ \eta_{TA}; \mu_A & = & id_{TA} & = T(\eta_A); \mu_A \end{array}$$

... *"a monad is a monoid in the category of endofunctors."*

But

...the monad notion in functional programming (applying to Haskell, too) lost its connection to the monad notion in philosophy and category theory (almost) completely, and hence, everything which might be or which might be considered a mystery or a miracle.

Rather than introducing a mystery, monads and monadic sequencing in functional programming close a 'functional gap' between function application, sequential function composition, and functorial mapping.

On the ‘Functional Gap’ and its Closing (1)

...smashing the myth behind functional programming monads.

- Function application (‘mapping over’):
  ```
  ($) :: (a -> b) -> a -> b
  g $ x = g x
  ```
  - Special case (m a for a, m b for b):
    ```
    ($) :: (m a -> m b) -> m a -> m b
    g $ x = g x
    ```
- Sequential function composition (‘sequencing’):
  ```
  (.) :: (b -> c) -> (a -> b) -> (a -> c)
  (f . g) x = f (g x)
  ```
  - Special case (m a for a, m b for b, m c for c):
    ```
    (.) :: (m b -> m c) -> (m a -> m b) -> (m a -> m c)
    (f . g) x = f (g x)
    ```

...one implementation fits all types: Parametric polymorphism

On the 'Functional Gap' and its Closing (2)

- Functorial mapping ('mapping over'):

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
fmap g c = ... '(unpack, map, pack)'

(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
(<*>) k c = ... '(unpack, unpack, map, pack)'
```

- Functorial mapping 2&3 (+ sequencing):

```
(fmap2) :: (Functor2 f) => (a -> f b) -> f a -> f b
(fmap2) k c  = ... '(unpack, map)'

(fmap3) :: (Functor3 f) => (f a -> b) -> f a -> f b
(fmap3) k c  = ... '(map, pack)'
```

- (Monadic) sequencing (+ mapping): Changing Focus

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>=) c k = fmap2 k c '(unpack, map, repeat >>=)'
```

...type-specific implementations fit every 1-ary type constructor:

Ad hoc polymorphism

Overall

...AdvancedSequencing, AdvancedComposing (or something alike) instead of Monad might have been a better choice for the class name for avoiding suggesting any mysthicism:

```
class AdvancedSequencing as where
 (>>=)  :: as a -> (a -> as b) -> as b
 return :: a -> as a
 (>>)   :: as a -> as b -> as b
 fail   :: String -> as a

 c >> k  =  c >>= \_ -> k
 fail s  =  error s

class AdvancedComposing ac where...
```

instead of the actually provided type constructor class Monad:

```
class Monad m where...
```

Chapter 11.8

References, Further Reading

Chapter 11: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17, Monaden)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 7, Monaden)
- Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis. University of Cambridge, British Computer Society Distinguished Dissertations in Computer Science, Cambridge University Press, 1992.

Chapter 11: Further Reading (2)

- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.2, The Monad Class; Chapter 18.3, The MonadPlus Class; Chapter 18.4, State Monads)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 10.6, Class and Instance Declarations – Monadic Types)
- John Launchbury, Simon Peyton Jones. *State in Haskell*. Lisp and Symbolic Computation 8(4):293-341, 1995.
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 13, A Fistful of Monads; Chapter 14, For a Few Monads More)

Chapter 11: Further Reading (3)

- Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-V., 1971 (2nd edition, 1998).
- Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
- Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
- Martin Odersky. *Funktionale Programmierung*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 599-612, 2006. (Kapitel 5.3, Funktionale Komposition: Monaden, Beispiele für Monaden)

Chapter 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)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 11, Beispiel: Monaden; Kapitel 17, Zeit und Zustand in der funktionalen Welt)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 21.2, Ein kommandobasiertes Ein-/Ausgabemodell; Kapitel 22.2, Kommandos; Kapitel 22.6.4, Anmerkungen zu Monaden)

Chapter 11: Further Reading (5)

- Thomas Petricek. *What We Talk about when We Talk about Monads*. The Art, Science, and Engineering of Programming 2(3), Article 12, 1-27, 2018.
- Simon Peyton Jones, John Launchbury. *State in Haskell*. Lisp and Symbolic Computation 8(4):293-341, 1995.
- Simon Peyton Jones, Philip Wadler. *Imperative Functional Programming*. In Conference Record of the 20 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 71-84, 1993.
- Simon Peyton Jones. *Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell*. In Tony Hoare, Manfred Broy, Ralf Steinbruggen (Eds.), Engineering Theories of Software Construction, IOS Press, 47-96, 2001 (Presented at the 2000 Marktoberdorf Summer School).

Chapter 11: Further Reading (6)

- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.2, Monads)
- Tom Schrijvers, Peter J. Stuckey, Philip Wadler. *Monadic Constraint Programming*. Journal of Functional Programming 19(6):663-697, 2009.
- Michael Spivey. *A Functional Theory of Exceptions*. Science of Computer Programming 14(1):25-42, 1990.
- Wouter S. Swierstra, Thorsten Altenkirch. *Beauty in the Beast: A Functional Semantics for the Awkward Sqad*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007), 25-36, 2007.

Chapter 11: Further Reading (7)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 18, Programming with actions; Chapter 18.8, Monads for functional programming)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 18, Programming with monads)
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.
- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.

Chapter 11: Further Reading (8)

- Philip Wadler. *Monads for Functional Programming*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 24-52, 1995.
- Philip Wadler. *How to Declare an Imperative*. In Proceedings of the 1995 International Symposium on Logic Programming (ILPS'95), Invited Presentation, MIT Press, 18-32, 1995.
- Philip Wadler. *How to Declare an Imperative*. ACM Computing Surveys 29(3):240-263, 1997.
- A (Reasonably) Comprehensive List of Tutorials on Monads: `haskell.org/haskellwiki/Monad_tutorials`.

Chapter 11: Monads - Background Reading (9)

- René Descartes. *Meditationes de prima philosophia*. 1641.
- Gottfried Wilhelm Leibniz. *Monadology* (Original in French). 90 Paragraphen, 1714.
- Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
- Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.
- Philip Wadler. *Monads for Functional Programming*. In Johan Jeuring, Erik Meijer (Eds.), *1st Int. Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 24-52, 1995.

Chapter 12

Arrows

Chapter 12.1

Motivation

Motivation

The higher-order type constructor class `Arrow`

- complements the type class `Monad`

with a complementary mechanism for

- function composition

which is amenable for 2-ary type constructors and useful e.g. for

- functional reactive programming (cf. Chapter 15).

Chapter 12.2

The Type Constructor Class Arrow

The Type Constructor Class Arrow

Arrows are 2-ary type constructors, which are instances of the type constructor class Arrows obeying the arrow laws:

```
class Arrow a where
 pure  :: (b -> c) -> a b c
     -- equivalently: pure :: ((->) b c) -> a b c
 (>>>) :: a b c -> a c d -> a b d
 first :: a b c -> a (b,d) (c,d)
```

Note:

- pure allows embedding of ordinary maps into the constructor class Arrow (the role of pure for maps is similar to the role of return in class Monad for values of type a).
- (>>>) serves the composition of computations.
- first has as an analogue on the level of ordinary functions the function firstfun with
  `firstfun f = \(x,y) -> (f x, y)`

The Arrow Laws

Proper instances of the the type constructor class `Arrow` must satisfy the following nine arrow laws:

Arrow Laws

```
pure id >>> f = f                              (AL1): identity
f >>> pure id = f                              (AL2): identity
(f >>> g) >>> h = f >>> (g >>> h)              (AL3): associa-
                                                      tivity

pure (g . f) = pure f >>> pure g               (AL4): functor
                                                      composition

first (pure f) = pure (f × id)                 (AL5): extension
first (f >>> g) = first f >>> first g          (AL6): functor
first f >>> pure (id × g) = pure (id × g) >>> first f
                                               (AL7): exchange
first f >>> pure fst = pure fst >>> f          (AL8): unit
first (first f) >>> pure assoc = pure assoc >>> first f
                                               (AL9): association
```

Instance (->) of Class Arrow (1)

...making the type constructor (->) an instance of the type constructor class Arrow:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f × id
```

where

```
(×) :: (b -> c) -> (d -> e) -> (b,d) -> (c,e)
(f × g)~(bv,dv) = (f bv, g dv) :: (c,e)
```

Note: Defining first by `first f = \(b,d) -> (f b, d)` would have been equivalent.

Instance (->) of Class Arrow (2)

...in more detail with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (->) an instance of Arrow means constructor a equals (->):

```
instance Arrow (->) where
```

$$\underbrace{\texttt{pure f}}_{\texttt{:: (->) b c}} \;\texttt{=}\; \underbrace{\texttt{f}}_{\texttt{:: (->) b c}}$$

$$\underbrace{\texttt{f}}_{\texttt{:: (->) b c}} \;\texttt{>>>}\; \underbrace{\texttt{g}}_{\texttt{:: (->) c d}} \;\texttt{=}\; \underbrace{\texttt{g . f}}_{\texttt{:: (->) b d}}$$

$$\underbrace{\texttt{first f}}_{\texttt{:: (->) b c}} \;\texttt{=}\; \underbrace{\texttt{f} \times \texttt{id}}_{\texttt{:: (->) (b,d) (c,d)}}$$

Recall: Defining `first` by `first f = \(b,d) -> (f b, d)` would have been equivalent.

Utility Functions (1)

The product map $\times$ (recalled):

```
(×) :: (a -> a') -> (b -> b') -> (a,b) -> (a',b')
(f × g)~(a,b) = (f a, g b)
```

Regrouping arguments via assoc, unassoc, and swap:

```
assoc :: ((a,b),c) -> (a,(b,c))
assoc~(~(x,y),z) = (x,(y,z))

unassoc :: (a,(b,c)) -> ((a,b),c)
unassoc~(x,~(y,z)) = ((x,y),z)

swap :: (a,b) -> (b,a)
swap~(x,y) = (y,x)
```

The dual analogue to the map first, the map second:

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = pure swap >>> first f >>> pure swap
```

Utitility Functions (2)

...derived operators for the type constructor class `Arrow`:

```
(***) :: Arrow a => a b c -> a b′ c′ ->
                                   a (b,b′) (c,c′)
f *** g = first f >>> second g

(&&&) :: Arrow a => a b c -> a b c′ -> a b (c,c′)
f &&& g = pure (_-> (b,b)) >>> (f *** g)

idA :: Arrow a => a b b
idA = pure id
```

Application: Modelling Circuits (1)

The map `add` introduces a notion of computation:

```
add :: (b -> Int) -> (b -> Int) -> (b -> Int)
add f g z = f z + g z
```

...which can be generalized in various ways.

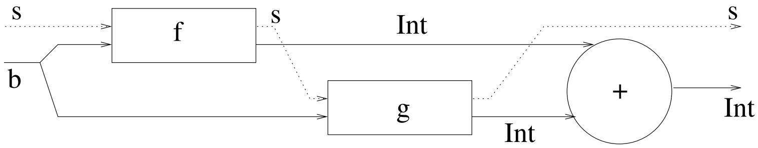
Application: Modelling Circuits (2)

First, generalizing add to state transformers:

```
type State s i o = (s,i) -> (s,o)

addST :: State s b Int -> State s b Int ->
                                    State s b Int
addST f g (s,z) = let (s′,x)  = f (s,z)
                      (s″,y) = g (s′,z)
                  in (s″,x+y)
```

Illustration:



Application: Modelling Circuits (3)

Second, generalizing add to non-determinism:

```
type NonDet i o = i -> [o]

addND :: NonDet b Int -> NonDet b Int ->
                                    NonDet b Int
addND f g z = [ x+y | x <- f z, y <- g z ]
```

Application: Modelling Circuits (4)

Third, generalizing add to map transformers:

```
type MapTrans s i o = (s -> i) -> (s -> o)

addMT :: MapTrans s b Int -> MapTrans s b Int ->
                                        MapTrans s b Int
addMT f g m z = f m z + g m z
```

Application: Modelling Circuits (5)

Fourth, generalizing add to simple automata:

```
newtype Auto i o = A (i -> (o, Auto i o))

addAuto :: Auto b Int -> Auto b Int -> Auto b Int
addAuto (A f) (A g)
   = A (\z -> let (x,f′) = f z
                  (y,g′) = g z
              in (x+y), addAuto f′ g′))
```

Putting all this together, it allows us

- modelling of synchronous circuits (with feedback loops).

Application: Modelling Circuits (6)

- Functions and programs often contain components that are 'function-like' 'w/out being just functions.'
- `Arrows` define a common interface for coping with the "notion of computation" of such function-like components.
- Monads are a special case of arrows.
- Like monads, arrows allow to meaningfully structure the computation process of programs.

Note

- The preceding examples have in common that there is a type $A \rightsquigarrow B$ of computations, where inputs of type A are transformed into outputs of type B.
- The type class `Arrow` yields a sufficiently general interface to describe these commonalities uniformly and to encapsulate them in a class.

Back to the Application

...next we are going to implement the previously introduced types as instances of the type constructor class `Arrow`. To this end, we reintroduce them as new types using `newtype`:

```
newtype State s i o = ST ((s,i) -> (s,o))

newtype NonDet i o  = ND (i -> [o])

newtype MapTrans s i o = MT ((s -> i) -> (s -> o))

newtype Auto i o = A (i -> (o, Auto i o))
```

Instance (State s) of Class Arrow (1)

...making state transformers an instance of Arrow:

```
newtype State s i o = ST ((s,i) -> (s,o))

instance Arrow (State s) where
  pure f          = ST (id × f)
  ST f >>> ST g = ST (g . f)
  first (ST f)  = ST (assoc . (f × id) . unassoc)
```

Instance (State s) of Class Arrow (2)

...in more detail with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (State s) an instance of Arrow means type constructor variable a is set to (State s):

```
newtype State s i o = ST ((s,i) -> (s,o))
```

instance Arrow (State s) where

$$\underbrace{\texttt{pure f}}_{\texttt{:: (->) b c}} \quad \texttt{=} \quad \underbrace{\texttt{ST (id} \times \texttt{f)}}_{\texttt{:: (State s) b c}}$$

$$\underbrace{\texttt{ST f}}_{\texttt{:: (State s) b c}} \quad \texttt{>>>} \quad \underbrace{\texttt{ST g}}_{\texttt{:: (State s) c d}} \quad \texttt{=} \quad \underbrace{\texttt{ST (g . f)}}_{\texttt{:: (State s) b d}}$$

$$\underbrace{\texttt{first (ST f)}}_{\texttt{:: (State s) b c}} \quad \texttt{=} \quad \underbrace{\texttt{ST (assoc . (f} \times \texttt{id) . unassoc)}}_{\texttt{:: (State s) (b,d) (c,d)}}$$

Instance NonDet of Class Arrow (1)

...making "non-determinism" an instance of Arrow:

```
newtype NonDet i o  = ND (i -> [o])

instance Arrow NonDet where
 pure f         = ND (\b -> [f b])
 ND f >>> ND g = ND (\b -> [d | c <- f b, d <- g c])
 first (ND f)   = ND (\(b,d) -> [(c,d) | c <- f b])
```

Instance NonDet of Class Arrow (2)

...in more detail with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making NonDet an instance of Arrow means type constructor variable a is set to NonDet:

```
NonDet i o  = ND (i -> [o])

instance Arrow NonDet where
  pure f          = ND (\b -> [f b])
  :: (->) b c        :: NonDet b c
     ND f    >>>    ND g      = ND (\b -> [d | c <- f b, d <- g c])
:: NonDet b c  :: NonDet c d                 :: NonDet b d
  first (ND f)       =   ND (\(b,d) -> [(c,d) | c <- f b])
     :: NonDet b c                :: NonDet (b,d) (c,d)
```

Instance (MapTrans s) of Class Arrow (1)

...Making map transformers an instance of Arrow:

```
newtype MapTrans s i o = MT ((s -> i) -> (s -> o))

instance Arrow (MapTrans s) where
 pure f         = MT (f .)
 MT f >>> MT g  = MT (g . f)
 first (MT f)   = MT (zipMap . (f x id) . unzipMap)
```

where

```
zipMap      :: (s -> a, s -> b) -> (s -> (a,b))
zipMap h s = (fst h s, snd h s)

unzipMap    :: (s -> (a,b)) -> (s -> a, s -> b)
unzipMap h = (fst . h, snd . h)
```

Instance (MapTrans s) of Class Arrow (2)

...in more detail with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (MapTrans s) an instance of Arrow means type constructor variable a is set to (MapTrans s):

```
MapTrans s i o = MT ((s -> i) -> (s -> o))
```

```
instance Arrow (MapTrans s) where
  pure f          =    MT (f .)
  :: (->) b c       :: (MapTrans s) b c
        MT f       >>>        MT g          =       MT (g . f)
 :: (MapTrans s) b c  :: (MapTrans s) c d  :: (MapTrans s) b d
  first (MT f)         =     MT (zipMap . (f x id) . unzipMap)
  :: (MapTrans s) b c          :: (MapTrans s) (b,d) (c,d)
```

Instance `Auto` of Class `Arrow` (1)

...Making simple automata an instance of `Arrow`:

```
newtype Auto i o = A (i -> (o, Auto i o))

instance Arrow Auto where
 pure f       = A (\b -> (f b, pure f))
 A f >>> A g = A (\b -> let (c,f′) = f b
                            (d,g′) = g c
                        in (d, f′ >>> g′)))
 first (A f) = A (\(b,d) -> let (c,f′) = f b
                            in ((c,d),first f′))
```

Instance Auto of Class Arrow (2)

...in more detail with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making Auto an instance of Arrow means type constructor variable a is set to Auto:

```
Auto i o = A (i -> (o, Auto i o))

instance Arrow Auto where
  pure f         = A (\b -> (f b, pure f)
  :: (->) b c            :: Auto b c
    A f    >>>    A g     = A (\b -> let (c,f') = f b
                                         (d,g') = g c
                                     in (d, f' >>> g')))
:: Auto b c  :: Auto c d              :: Auto b d
  first (A f)     =    A (\(b,d) -> let (c,f') = f b
                                     in ((c,d),first f'))
     :: Auto b c               :: Auto (b,d) (c,d)
```

Contents
Part I
Chap. 1
Part II
Chap. 2
Chap. 3
Chap. 4
Part III
Chap. 5
Chap. 6
Part IV
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3

Last but not least

...generalization:

Consider the general combinator:

```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> pure (uncurry (+))
```

Note: Each of the considered variants of add results as a specialization of addA with the corresponding arrow-type.

Summing up

- Arrow-combinators operate on 'computations', not on values. They are point-free in distinction to the 'common case' of functional programming.
- Analoguous to the monadic case a `do`-like notational variant makes programming with arrow-operations often easier and more suggestive (cf. literature hint at the end of the chapter), whereas the pointfree variant is more useful and advantageous for proof-theoretic reasoning.

Last but not least (1)

...compare (same color means "correspond to each other"):

```
(.) :: (b -> c) -> (a -> b ) -> (a -> c)
(f . g) v = f (g v)

(;) :: (a -> b ) -> (b -> c) -> (a -> c)
(f ; g) = g . f                               -- pointfree

(>>;) :: a -> (a -> b) -> b
v >>; f = f v

(;<<) :: (a -> b) -> a -> b
f ;<< v = v >>; f                  -- Non-monadic operations
```

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b -- Monadic op.
f =<< x = x >>= f

(>>=) :: Monad m => m a -> (a -> m b) -> m b
m >>= k = k v...                               -- "m = dc v"

(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >@> g = \x -> (f x) >>= g

(<@<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
f <@< g = g >@> f                              -- pointfree
```

Last but not least (2)

```
(>>>) :: Arrow a => a b c -> a c d -> a b d
```

...introduces composition for 2-ary type constructors.

Reconsider now instance `(->)` of class `Arrow`:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f × id
```

This means: For `(->)` as `Arrow` instance

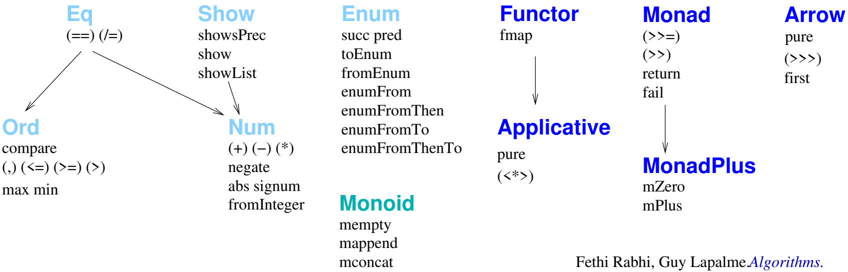
- arrow composition boils down to ordinary function composition, i.e.: `(>>>) = (.)`

Chapter 12.3

A Fresh Look at the Haskell Class Hierarchy

Monoids, Monads, Functors, Arrows,...

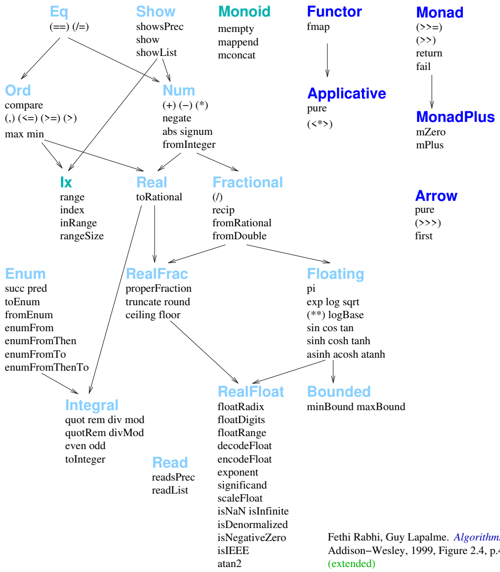
...as part of the Haskell'98 type class hierarchy:



Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison–Wesley, 1999, Figure 2.4, p.46 (extended)

Type Classes and Type Class Functions

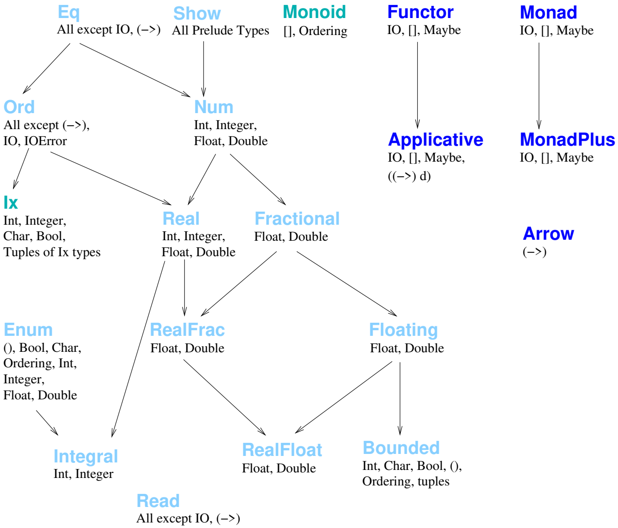
...of a section of the Haskell'98 type class hierarchy:



Fethi Rabhi, Guy Lapalme. *Algorithms.*
Addison−Wesley, 1999, Figure 2.4, p.46
(extended)

Type Classes and Type Class Instances

...of a section of the Haskell'98 type class hierarchy:



Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000, p.156 (extended)

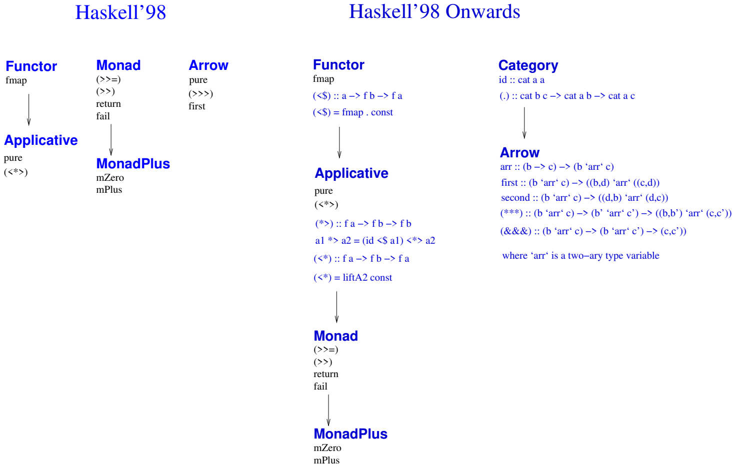
'98 Type Class Memberships of Selected Types

Type	Instance of	Derivation
()	Read	Eq Ord Enum Bounded
[a]	Read	Eq Ord
[]	Functor Applicative Monad MonadPlus	
(a,b)	Read	Eq Ord Bounded
((−>) d)	Functor Applicative	
(−>)	Arrow	
Array	Functor Eq Ord Read	
Bool		Eq Ord Enum Read Bounded
Char	Eq Ord Enum Read	
Complex	Floating Read	
Double	RealFloat Read	
Either		Eq Ord Read
Float	RealFloat Read	
Int	Integral Bounded Ix Read	
Integer	Integral Ix Read	
IO	Functor Applicative Monad MonadPlus	
IOError	Eq	
Maybe	Functor Applicative Monad MonadPlus	Eq Ord Read
Ordering	Monoid	Eq Ord Enum Read Bounded
Ratio	RealFrac Read	

Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison−Wesley, 1999, Table 2.4, p. 47 (extended)

Update on the Haskell Type Class Hierarchy

...Haskell is a research vehicle and, hence, a moving target:



...for more information, check out:

https://wiki.haskell.org/Typeclassopedia

Chapter 12.4

References, Further Reading

Chapter 12: Further Reading

- Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.
- John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37:67-111, 2000.
- Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001), 229-240, 2001.
- Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 201-222, 2003.

Es ist nicht genug zu wissen,
man muss auch anwenden.

Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Part V

Applications

Chapter 13

Parsing

Parsing: Lexical and Syntactical Analysis

Parsing

- a common term for the lexical and syntactical analysis of the structure of text, e.g., source code text of programs.
- an(other) application often used for demonstrating the power and elegance of functional programming.
- enjoys a long history, see e.g.
  - William H. Burge. Recursive Programming Techniques. Addison-Wesley, 1975.

  as an example of an early text book concerned with parsing.

Functional Approaches for Parsing

...two different but conceptually related approaches are:

- Combinator parsing
  - Graham Hutton. Higher-Order Functions for Parsing. Journal of Functional Programming 2(3):323-343, 1992.
- Monadic parsing
  - Graham Hutton, Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
  - Graham Hutton, Erik Meijer. Monadic Parsing in Haskell. Journal of Functional Programming 8(4):437-444, 1998.

which are both well-suited for building recursive descent parsers.

Chapter 13.1

Motivation

Informally

...the parsing problem is as follows:

- Read a sequence of objects of some type a.
- Yield an object or a sequence of objects of some type b.

Example: Reading a sequence of objects of type Char:

⟨ `if n mod = 0 then 2*n else 2*n+1 fi` ⟩

Yielding a sequence of objects of (enriched symbol) tokens:

```
⟨ (if_symb,""),(var_symb,"n"),(op_symb,"mod"),
  (rel_symb,"="),(cst_symb,"0"),(then_symb,""),
  (cst_symb,"2"),(op_symb,"*"),(var_symb,"n"),
  (else_symb,""),...,(fi_symb,"") ⟩
```

Parsing Arithmetic Expressions

...a more complex parsing problem: Write a parser *p*, which

- reads a string *s* representing a well-formed arithmetic expression (e.g., `s = "((2+b)*5)"`)
- yields the value of type `Exp` represented by *s* with:

```
data Exp = Lit Int | Var Char | Op Ops Exp Exp
data Ops = Add | Sub | Mul | Div | Mod
```

Applied to string `"((2+b)*5)"`, e.g., parser *p* shall deliver the `Exp`-value:

```
Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
```

Note: *p* can be considered the reverse of the `show` function. It is also similar to the automatically derived `read` function for `Expr`: *p* and `read`, however, differ in the arguments they accept: strings of the compact form `"((2+b)*5)"` vs. strings of the form `"Op Mul (Add (Lit 2) (Var 'b')) (Lit 5)"`.

Towards the Type of Parser Functions (1)

...characterizing parsing as the

- reading of sequences s of objects of some type a
- yielding objects or lists of objects of some type b

suggests naively for the type of parser functions:

```
type NaiveParse a b = [a] -> b
```

This, however, raises some questions. Assume, bracket and number are parser functions for detecting brackets and numbers, respectively:

```
Parser    Input        What shall be the output?
bracket   "(xyz"  ->> '('? If so, what to do w/ "xyz"?
number    "234"   ->> 2? Or 23? Or 234?
bracket   "234"   ->> No result? Failure?
```

Towards the Type of a Parser Function (2)

In detail: How shall a parser function behave if

- the input is not completely read?
- there are multiple results?
- there is a failure?

Answering the latter two questions first suggests to refine the type of parser functions to:

```
type RefinedParse a b = [a] -> [b]
```

which allows the following parsing output for the previous example:

```
Parser    Input      Expected Output
bracket   "(xyz"  ->> ['(']
number    "234"   ->> [2,23,234]
bracket   "234"   ->> []
```

Towards the Type of a Parser Function (3)

The first question, however, has still to be answered:

- What shall a parser function do with the part of the input that has not been read?

Answering it leads to the definite definition of the type of parser functions:

```
type Parse a b = [a] -> [(b,[a])]
```

...enabling the parsing output:

```
Parser  Input      Output
bracket "(xyz" ->> [('(',"xyz")]
number  "234"  ->> [(2,"34"), (23,"4"), (234,"")]
bracket "234"  ->> []
```

Intuitively

...if a parser function delivers

- the empty list, this signals failure of the analysis.
- a non-empty list, this signals success of the analysis: Every list element represents the result of a successful parse.

In the success case, every list element is a pair, whose

- first component is the identified object (token)
- second component is the remaining input which still needs to be analyzed.

Note, using lists for enabling the delivery of multiple results

- is known as the so-called list of successes technique (Philip Wadler, 1985).
- enables parsers to analyze also ambiguous grammars.

Note

...the following presentation is based on:

- Simon Thompson. Haskell – The Craft of Functional Programming, Addison-Wesley/Pearson, 2nd edition, 1999, Chapter 17.
- Graham Hutton, Erik Meijer. Monadic Parsing in Haskell. Journal of Functional Programming 8(4):437-444, 1998.

Chapter 13.2

Combinator Parsing

Objective

...developing a combinator library for parsing composed of

- Four primitive parser functions
  - Two of which are input-independent (`none`, `succeed`)
  - Two of wich are input-dependent (`token`, `spot`)
- Three parser combinators for
  - Alternatives (`alt`)
  - Sequencing (`(>*>)`)
  - Transforming (`build`)

...forming a universal parser basis, which allows to construct parser functions at will, i.e., according to what is required by a parsing problem.

Chapter 13.2.1

Primitive Parsers

The two Input-independent Primitive Parsers

Recall:

```
type Parse a b = [a] -> [(b,[a])]
```

- `none`, the always failing parser

  ```
  none :: Parse a b
  none _ = []
  ```

- `succeed`, the always succeeding parser

  ```
  succeed :: b -> Parse a b
  succeed val inp = [(val,inp)]
  ```

Note:

- Parser `none` always fails. It does not accept anything.
- Parser `succeed` always succeeds without consuming its input or parts of it. In BNF-notation this corresponds to the symbol ε representing the empty word.

The two Input-dependent Primitive Parsers

- token, the parser recognizing single objects (so-called tokens):

```
token :: Eq a => a -> Parse a a
token t (x:xs)
 | t == x      =  [(t,xs)]
 | otherwise   =  []
token t []     =  []
```

- spot, the parser recognizing single objects enjoying some property:

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
 | p x         =  [(x,xs)]
 | otherwise   =  []
spot p []      =  []
```

Example: Using the Primitive Parsers

...for constructing parsers for simple parsing problems:

```
bracket = token '('
dig     = spot isDigit

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

Note: The parser functions `token` and `bracket` could also be defined using `spot`:

```
token :: Eq a => a -> Parse a a
token t = spot (== t)

bracket :: Char -> Parse Char Char
bracket = spot (== '(')
```

Chapter 13.2.2

Parser Combinators

Parser Combinators

...to write more complex and powerful parser functions, we need in addition to primitive parsers

- parser-combining functions (or parser combinators)

which are re-usable higher-order polymorphic functions.

The Parser Combinator for Comb. Alternatives

Combining parsers as alternatives:

- alt, the parser combining parsers as alternatives:

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

Intuitively: alt combines the results of the parses of p1 and p2. The success of either of them is a success of the combination.

Example:

```
(bracket `alt` dig) "234" ->> [] ++ [(2,"34")]
                          ->> [(2,"34")]
```

More generally, an expression is either a literal, or a variable or an operator expression:

```
(lit `alt` var `alt` opexp) "(234+7)" ->> ...
```

The Parser Combinator for Sequential Comp.

Combining parsers sequentially:

- (>*>), the parser combining parsers sequentially:

```
infixr 5 >*>
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp,
                    (z,rem2) <- p2 rem1 ]
```

Intuitively

- The values (y,rem1) run through the results of parser p1 applied to inp. For each of them, parser p2 is applied to rem1, the unconsumed part of the input by p1 in that particular case. The results of the two successful parses, y and z, are returned as a pair.

  E.g., an operator expression starts with a bracket (detected by parser bracket) followed by a number (detected by parser number).

Example for Sequentially Composing Parsers

...evaluating `number "24("` yields a list of two parse results `[(2,"4("), (24,"(")]`. We thus get for the composition of the parsers `number` and `bracket` applied to input `"24("`:

```
(number >*> bracket) "24("
 ->> [((y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"(")],
                     (z,rem2) <- bracket rem1 ]
 ->> [((2,z),rem2) | (z,rem2) <- bracket "4(" ] ++
     [((24,z),rem2) | (z,rem2) <- bracket "(" ]
 ->> [] ++ [((24,z),rem2) | (z,rem2) <- bracket "(" ]
 ->> [((24,z),rem2) | (z,rem2) <- bracket "(" ]
 ->> [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
 ->> [((24,'('), "")]
```

The Parser Combinator for Transformations

Combining a parser with a map transforming the parse results:

- build, the parser transforming obtained parse results:

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [(f x,rem) | (x,rem) <- p inp]
```

Intuitively: The map argument f of build transforms the items returned by its parser argument: It builds something from it.

Example for Transforming Parse Results

...the parser digList is assumed to return a list of digit lists, whose elements are transformed by digsToNum into the numbers whose values they represent:

```
(digList `build` digsToNum) "21a3"
 ->> [(digsToNum x,rem) | (x,rem) <- digList "21a3"]
 ->> [(digsToNum x,rem) | (x,rem) <-
                              [("2","1a3"),("21","a3")]]
 ->> [(digsToNum "2","1a3"),(digsToNum "21","a3")]
 ->> [(2,"1a3"),(21,"a3")]
```

Chapter 13.2.3

Universal Combinator Parser Basis

Universal Combinator Parser Basis

...together, the four primitive parsers

- `none`, `succeed`, `token`, and `spot`

and the three parser combinators

- `alt`, `(>*>)`, and `build`

form a universal combinator parser basis, i.e., they allow us to build any parser we might be interested in.

The Universal Parser Basis at a Glance (1)

The priority of the sequencing operator:

```
infixr 5 >*>
```

The type of parser functions:

```
type Parse a b = [a] -> [(b,[a])]
```

Two input-independent primitive parser functions:

- The always failing parser function:

  ```
  none :: Parse a b
  none _ = []
  ```

- The always succeeding parser function:

  ```
  succeed :: b -> Parse a b
  succeed val inp = [(val,inp)]
  ```

The Universal Parser Basis at a Glance (2)

Two input-dependent primitive parser functions:

- The parser for recognizing single objects:

```
token :: Eq a => a -> Parse a a
token t = spot (==t)
```

- The parser for recognizing single objects satisfying some property:

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
 | p x          =  [(x,xs)]
 | otherwise    =  []
spot p []       =  []
```

The Universal Parser Basis at a Glance (3)

Three parser combinators:

- Alternatives

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

- Sequencing

```
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp,
                    (z,rem2) <- p2 rem1]
```

- Transformation

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [(f x, rem) | (x,rem) <- p inp]
```

Chapter 13.2.4

Structure of Combinator Parsers

The Structure of Combinator Parsers

...is usually as follows:

```
type Parse a b = [a] -> [(b,[a])]

none     :: Parse a b
succeed  :: b -> Parse a b
token    :: Eq a => a -> Parse a a
spot     :: (a -> Bool) -> Parse a a
alt      :: Parse a b -> Parse a b -> Parse a b
(>*>)    :: Parse a b -> Parse a c -> Parse a (b,c)
build    :: Parse a b -> (b -> c) -> Parse a c
list     :: Parse a b -> Parse a [b]
topLevel :: Parse a b -> [a] -> b    -- see Exam. 2,
                                     -- Chap. 13.2.5
```

Combinator Parsers

...are well-suited for writing so-called recursive descent parsers.

This is because the parser functions (summarized on the previous slide)

- are structurally similar to grammars in BNF-form.
- provide for every operator of the BNF-grammar a corresponding (higher-order) parser function.

These (higher-order) parser functions allow

- combining simple(r) parsers to (more) complex ones.
- are therefore called combining forms, or, as a short hand, combinators (cf. Graham Hutton. Higher-order Functions for Parsing. Journal of Functional Programming 2(3), 323-343, 1992).

Chapter 13.2.5

Writing Combinator Parsers: Examples

Using the Parser Basis

...for constructing (more) complex parser functions.

A parser

- recognizing a list of objects (example 1).
- transforming a string expression into a value of a suitable algebraic data type for expressions (example 2).

Example 1: Parsing a List of Objects

...let p be a parser recognizing single objects. Then list applied to p is a parser recognizing lists of objects:

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) `alt`
         ((p >*> list p) `build` (uncurry (:)))
```

Intuitively

- A list of objects can be empty: This is recognized by the parser `succeed` called with `[]`.
- A list of objects can be non-empty, i.e., it consists of an object followed by a list of objects: This is recognized by the sequentially composed parsers `p` and `(list p)`: `(p >*> list p)`.
- The parser `build`, finally, is used to turn a pair `(x,xs)` into the list `(x:xs)`.

Example 2: Parsing String Expressions (1)

...back to the initial example: Parsing string expressions like `"(234+∼42)*b"`, we shall construct the corresponding value of the algebraic data type:

```
data Expr = Lit Int | Var Char | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod
```

Parsing `"(234+∼42)*b"`, e.g., shall yield the `Exp`-value:

```
Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')
```

...according to the below assumptions for string expressions:

- Variables are the lower case characters from 'a' to 'z'.
- Literals are of the form 67, ∼89, etc., where ∼ is used for unary minus.
- Binary operators are $+, *, -, /, \%$, where / and % represent integer division and modulo operation, respectively.
- Expressions are fully bracketed.
- White space is not permitted.

Example 2: Parsing String Expressions (2)

The parser for string expressions:

```
parser :: Parse Char Expr
parser = nameParse `alt` litParse `alt` opExpParse
```

...is composed of three parsers reflecting the three kinds of expressions:

- variables (or variable names)
- literals (or numerals)
- fully bracketed operator expressions.

Example 2: Parsing String Expressions (3)

Parsing variable names:

```
nameParse :: Parse Char Expr
nameParse = spot isName `build` Var

isName :: Char -> Bool                   -- A variable name
isName x = ('a' <= x && x <= 'z')        -- must be a lower
                                         -- case character
```

Parsing literals (numerals):

```
litParse :: Parse Char Expr
litParse                            -- A literal starts
 = ((optional (token '~')) >*>      -- optionally with '~'
    (neList (spot isDigit))         -- followed by a non-
   `build` (charlistToExpr . uncurry (++))) -- empty
                                    -- list of digits
```

Example 2: Parsing String Expressions (4)

Parsing fully bracketed operator expressions:

```
optExpParse :: Parse Char Expr
opExpParse           -- A non-trivial expression
 = (token '(' >*>    -- must start with an opening bracket,
    parser    >*> -- must be followed by an expression,
    spot isOp >*> -- must be followed by an operator,
    parser    >*> -- must be followed by an expression,
    token ')')       -- must end with a closing bracket.
    `build` makeExpr
```

Example 2: Parsing String Expressions (4)

...required supporting parser functions:

```
neList   :: Parse a b -> Parse a [b]
optional :: Parse a b -> Parse a [b]
```

where

- `neList p` recognizes a non-empty list of the objects recognized by `p`.
- `optional p` recognizes an object recognized by `p` or succeeds immediately.

Note: `neList`, `optional`, and some other supporting functions including

- `isOp`
- `charlistToExpr`

are still be defined, which is left here as homework.

Example 2: Parsing String Expressions (5)

...we are left with defining a top-level parser function, which converts a string into an expression when called with `parser`:

Converting a string into the expression it represents:

```
topLevel :: Parse a b -> [a] -> b
topLevel p input
  = case results of
      [] -> error "parse unsuccessful"
      _  -> head results
    where
     results = [found | (found, []) <- p input]
```

Note:

- The parse of an input is successful, if the result contains at least one parse, in which all the input has been read.
- `topLevel parser "(234+~42)*b)" ->>`
  `Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')`

Chapter 13.3

Monadic Parsing

Monadic Parsing

...complements the concept of combining forms underlying combinator parsing with the one of monads.

For rendering this possible, the type of parser functions needs to be adjusted in order to make it a 1-ary type constructor which is eligible as an instance of type class Monad:

```
newtype Parser a = Parse (String -> [(a,String)])
```

while re-using the convention of Chapter 13.2 that delivery of the

- empty list signals failure of a parsing analysis.
- non-empty list signals success of a parsing analysis: each element of the list is a pair, whose first component is the identified object (token) and whose second component the input which is still to be parsed.

Chapter 13.3.1

Parser as Monads

Making Parser an Instance of Monad

Recalling the definition of type class Monad:

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b -- (>>), failure are not
 return :: a -> m a                 -- not needed: Their de-
                                    -- fault implement. apply.
```

...Parser, a 1-ary type constructor, is made an instance of Monad as follows:

```
instance Monad Parser where
 p >>= f  = Parse (\cs -> concat [(parse (f a)) cs′ |
                            (a,cs′) <- (parse p) cs])
 return a = Parse (\cs -> [(a,cs)])
```

where

```
parse :: (Parser a) -> (String -> [(a,String)])
parse (Parse p) = p
```

Remarks on Parser as an Instance of Monad

```
instance Monad Parser where
 p >>= f  = Parse (\cs -> concat [(parse (f a)) cs′ |
                              (a,cs′) <- (parse p) cs])
 return a = Parse (\cs -> [(a,cs)])
```

Intuitively:

- The parser `(return a)` succeeds without consuming any of the argument string, and returns the single value `a`.
- `parse` denotes a deconstructor function for parsers defined by `parse (Parse p) = p`.
- The parser sequence `p >>= f` applies first parser `(parse p)` to the argument string `cs` yielding a list of results of the form `(a,cs′)`, where `a` is a value and `cs′` is a string. For each such pair the parser `(parse (f a))` is applied to the unconsumed input string `cs′`. The result is a list of lists which is concatenated to give the final list of results.

Proof Obligations for `Parser` as a `Monad` Inst.

...we can prove that `Parser` satisfies the monad laws and is thus a valid instance of `Monad`:

Lemma 13.3.1.1 (Monad Laws)

```
          return a >>= f = f a
               p >>= return = p
p >>= (\a -> (f a >>= g)) = (p >>= (\a -> f a)) >>= g
```

Note:

- `(>>=)` being associative allows suppression of parentheses when parsers are applied sequentially.
- `return` being left-unit and right-unit for `(>>=)` allows some parser definitions to be simplified.

Chapter 13.3.2

Parsers by Type Class Instantiations

Note

...having made `Parser` an instance of `Monad` provides us with two important parser functions, a primitive parser and a (monadic) parser combinator:

- `return`, the always succeeding parser
- `(>>=)`, a combinator for sequentially combining parsers

which are the monadic counterparts of the parser combinators

- `succeed`
- `(>*>)`

of Chapter 13.2.1 and 13.2.2, respectively.

Making `Parser` an instance of `MonadPlus` will provide us with two further parser functions...

Making Parser an Instance of MonadPlus

...where MonadPlus is defined by (cf. Chapter 11.6):

```
class Monad m => MonadPlus m where
 mzero :: m a
 mplus :: m a -> m a -> m a
```

will provide us with the parser functions:

- mzero, the always failing parser
- mplus (via (++)), the parser for alternatives (or non-deterministic choice)

which are the monadic counterparts of the parser combinators

- none
- alt

of Chapter 13.2.1 and 13.2.2, respectively.

The Instance Decl. of Parser for MonadPlus

...yields the new parser functions mzero and mplus:

```
instance MonadPlus Parser where
 -- The always failing parser
 mzero = Parse (\cs -> [])

 -- The parser combinator for alternatives:
 p `mplus` q = Parse (\cs -> parse p cs ++ parse q cs)
```

Note: mplus can yield more than one result; the value of (parse p cs ++ parse q cs) can be a list of any length. In this sense mplus is considered to explore parsers alternatively (or, in this sense, non-deterministically).

Proof Obligat. for Parser as MonadPlus Inst.

...we can prove that Parser satisfies the MonadPlus laws:

Lemma 13.3.2.1 (MonadPlus Laws)

```
p >>= (\_ -> mzero) = mzero
mzero >>= p         = mzero
mzero `mplus` p     = p
p `mplus` mzero     = p
```

Intuitively, this means:

- mzero is left-zero and right-zero for (>>=).
- mzero is left-unit and right-unit for mplus.

Moreover

...we can prove the following laws:

Lemma 13.3.2.2

```
       p `mplus` (q `mplus` r) = (p `mplus` q) `mplus` r
           (p `mplus` q) >>= f = (p >>= f) `mplus` (q >>= f)
p >>= (\a -> f a `mplus` g a) = (p >>= f) `mplus` (p >>= g)
```

Intuitively, this means:

- `mplus` is associative.
- `(>>=)` distributes through `mplus`.

Chapter 13.3.3

Universal Monadic Parser Basis

Note

…in order to arrive at a universal monadic parser basis as in Chapter 13.2.3 we are left with defining monadic counterparts of the

- primitive parsers `token` and `spot`.
- parser combinator `build`.

The Monadic Counterpart of Parser spot

...parser sat recognizing single characters satisfying a given property:

```
sat  :: (Char -> Bool) -> Parser Char
sat p =
 do {c <- item; if p c then return c else zero}
```

is the monadic counterpart of the parser function token of Chapter 13.2.1.

The Monadic Counterpart of Parser token

...parser char recognizing single characters defined in terms of parser sat:

```
char   :: Char -> Parser Char
char c = sat (== c)
```

is the monadic counterpart of the parser function token of Chapter 13.2.1.

The Universal Monadic Parser Basis (1)

The type of parser functions:

```
newtype Parser a = Parse (String -> [(a,String)])
```

Two input-independent primitive parser functions:

- The always failing parser function:

```
mzero :: Parser a
mzero = Parse (\cs -> [])
```

- The always succeeding parser function:

```
return :: a -> Parser a
return a = Parse (\cs -> [(a,cs)])
```

The Universal Monadic Parser Basis (2)

Two input-dependent primitive parser functions:

- The parser for recognizing single objects:

```
char :: Char -> Parser Char
char c = sat (== c)
```

- The parser for recognizing single objects satisfying some property:

```
sat :: (Char -> Bool) -> Parser Char
sat p =
  do {c <- item; if p c then return c else zero}
```

The Universal Monadic Parser Basis (3)

Three parser combinators:

- Alternatives

```
mplus :: Parser a -> Parser a -> Parser a
p `mplus` q =
  Parse (\cs -> parse p cs ++ parse q cs)
```

- Sequencing

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= f =
 Parse (\cs -> concat [(parse (f a)) cs' |
                          (a,cs') <- (parse p) cs])
```

- Transformation

```
mbuild :: Parser a -> (a -> b) -> Parser b
mbuild p f inp = ... (completion left as homework)
```

Chapter 13.3.4

Utility Parsers

Utility Parsers (1)

Consuming the first character of an input string, if it is non-empty, and failing otherwise:

```
item :: Parser Char
item  = Parse (\cs -> case cs of
                        ""     -> []
                        (c:cs) -> [(c,cs)])
```

Parsing a specific string:

```
string :: String -> Parser String
string ""     = return ""
string (c:cs) = do char c; string cs; return (c:cs)
```

Utility Parsers (2)

The deterministically selecting parser:

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q
 = Parse (\cs -> case parse (p `mplus` q) cs of
                    []     -> []
                    (x:xs) -> [x])
```

Note:

- `(+++)` shows the same behavior as `mplus`, but yields at most one result (in this sense 'deterministically'), whereas `mplus` can yield several ones (in this sense 'non-deterministically')
- `(+++)` satisfies all of the previously listed properties of `mplus`.

Utility Parsers (3)

Applying a parser p repeatedly:

```
-- zero or more applications of p
many :: Parser a -> Parser [a]
many p = many1 p +++ return []

-- one or more applications of p
many1 :: Parser a -> Parser [a]
many1 p = do a <- p; as <- many p; return (a:as)
```

Note: As above, useful parsers are often recursively defined.

Utility Parsers (4)

A variant of the parser many with interspersed applications of parser sep, whose result values are thrown away:

```
sepby :: Parser a -> Parser b -> Parser [a]
p `sepby` sep = (p `sepby1` sep) +++ return []

sepby1 :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do a  <- p
                    as <- many (do sep; p)
                    return (a:as)
```

Utility Parsers (5)

Repeated applications of a parser p separated by applications of a parser op, whose result value is an operator which is assumed to associate to the left, and used to combine the results from the p parsers in chainl and chainl1:

```
chainl :: Parser a -> Parser (a -> a -> a)
                                   -> a -> Parser a
chainl p op a = (p `chainl1` op) +++ return a

chainl1 :: Parser a -> Parser (a -> a -> a)
                                    -> Parser a
p `chainl1` op = do a <- p; rest a
                 where rest a = (do f <- op
                                    b <- p
                                    rest (f a b))
                                +++ return a
```

Utility Parsers (6)

Handling white space, tabs, newlines, etc.

- Parsing a string with blanks, tabs, and newlines:

```
space :: Parser String
space  = many (sat isSpace)
```

- Parsing a token by means of a parser p skipping any ‘trailing’ space:

```
token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}
```

- Parsing a symbolic token:

```
symb   :: String -> Parser String
symb cs = token (string cs)
```

- Applying a parser p and throwing away any leading space:

```
apply  :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p})
```

Note

...parsers handling comments or keywords can be defined in a similar fashion allowing together avoidance of a dedicated lexical analysis (for token recognition), which typically precedes parsing.

Chapter 13.3.5

Structure of a Monadic Parser

The Typical Structure of a Monadic Parser

...using the sequencing operator `(>>=)` or the syntactically sugared `do`-notation:

```
p1 >>= \a1 ->              do a1 <- p1
p2 >>= \a2 ->                 a2 <- p2
...                           ...
pn >>= \an ->                 an <- pn
f a1 a2 ... an                f a1 a2 ... an
```

...the latter one equivalently expressed in just one line, if so desired:

```
do {a1 <- p1; a2 <- p2;...; an <- pn; f a1 a2...an}
```

Recall: The expressions `ai <- pi` are called generators (since they generate values for the variables `ai`). Generators of the form `ai <- pi` can be replaced by `pi`, if the generated value will not be used afterwards.

Note

...the intuitive, natural operational reading of such a monadic parser:

- Apply parser `p1` and call its result value `a1`.
- Apply subsequently parser `p2` and call its result value `a2`.
- ...
- Apply subsequently parser `pn` and call its result value `an`.
- Combine finally the intermediate results by applying an appropiate function `f`.

Note, most typically `f = return (g a1 a2 ... an)`; for an exception see parser `chainl1` in Chapter 13.3.4, which needs to parse 'more of the argument string' before it can return a result.

Chapter 13.3.6

Writing Monadic Parsers: Examples

Example 1: A Simple Parser

...writing a parser p which

- reads three characters,
- drops the second character of these, and
- returns the first and the third character as a pair.

Implementation:

```
p :: Parser (Char,Char)
p = do c <- item; item; d <- item; return (c,d)
```

Example 2: Parsing Arithm. Expressions (1)

...built up from single digits, the operators +, -, *, /, and parentheses, respecting the usual precedence rules for additive and multiplicative operators.

Grammar for arithmetic expressions:

```
expr   ::=  expr addop term | term
term   ::=  term mulop factor | factor
factor ::=  digit | (expr)
digit  ::=  0 | 1 | ... | 9

addop  ::=  + | -
mulop  ::=  * | /
```

Example 2: Parsing Arithm. Expressions (2)

The Parsing Problem:

Parsing expressions and evaluating them on-the-fly (yielding their integer values) using the `chainl1` combinator of Chapter 13.3.4 to implement the left-recursive production rules for `expr` and `term`.

Example 2: Parsing Arithm. Expressions (3)

The implementation of the parser expr:

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr = term   `chainl1` addop
term = factor `chainl1` mulop

factor =
 digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit =
 do {x <- token (sat isDIgit); return (ord x - ord `0`)}

addop = do {symb "+"; return (+)}
        +++ do {symb "-"; return (-)}

mulop = do {symb "*"; return (*)}
        +++ do {symb "/"; return (div)}
```

Example 2: Parsing Arithm. Expressions (4)

...using the parser.

Parsing and evaluating the string `" 1 - 2 * 3 + 4 "` on-the-fly by calling:

```
apply expr " 1 - 2 * 3 + 4 "
```

yields the singleton list:

```
[(-1,"")]
```

which is the desired result.

Chapter 13.4

Summary

In conclusion

...non-monadic and monadic parsing rely (in part) on different language features but are quite similar in spirit as becomes obvious when opposing their primitives and combinators:

	Combinator Parsing	Monadic Parsing

Primitive Parsers	`none`
	`succeed`
	`token`
	`spot`
		`mzero`
		`return`
		`char`
		`sat`

Parser Combinators	`alt`
	`(>*>)`
	`build`
		`mplus`
		`(>>=)`
		`mbuild`

Invaluable

...for combinator (as well as monadic) parsing are:

- Higher-order functions: `Parse a b` (like `Parser a`) is of a functional type; all parser combinators are thus higher-order functions.
- Polymorphism: The type `Parse a b` is polymorphic: We do need to be specific about either the input or the output type of the parsers we build. Hence, the parser combinators mentioned above can immediately be reused for tokens of any other data type (in the examples, these were lists and pairs, characters, and expressions).
- Lazy evaluation: 'On demand' generation of the possible parses, automatical backtracking (the parsers will backtrack through the different options until a successful one is found).

Chapter 13.5

References, Further Reading

Chapter 13.2: Further Reading (1)

- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2nd edition, 1998. (Chapter 11, Parsing)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 13.1.3, Ausdrücke parsen; Kapitel B.1, Der Quellcode für den Ausdrucksparser)
- Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010. (Chapter 9, Parsing)
- Jeroen Fokker. *Functional Parsers*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 1-23, 1995.

Chapter 13.2: Further Reading (2)

- Steve Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6(3):445-464, 1996.
- Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
- Pieter W.M. Koopman, Marinus J. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), Selected Papers, Springer-V., LNCS 1595, 120-136, 1999.
- Matthew Might, David Darais, Daniel Spiewak. *Parsing with Derivatives – A Functional Pearl*. In Proceedings of the 16th ACM International Conference on Functional Programming (ICFP 2011), 189-195, 2011.

Chapter 13.2: Further Reading (3)

- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 18.6.2, Ausdrücke als Bäume)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 3, Parser als Funktionen höherer Ordnung)
- S. Doaitse Swierstra. *Combinator Parsing: A Short Tutorial*. In Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Revised Tutorial Lectures. Springer-V., LNCS 5520, 252-300, 2009.

Chapter 13.2: Further Reading (4)

- S. Doaitse Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer-V., LNCS 1725, 111-129, 1999.
- S. Doaitse Swierstra, Luc Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer-V., LNCS 1129, 184-207, 1996.
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 17.5, Case study: parsing expressions)

Chapter 13.2: Further Reading (5)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17.5, Case study: parsing expressions)
- Philip Wadler. *How to Replace Failure with a List of Successes*. In Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 113-128, 1985.

Chapter 13.3: Further Reading (6)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer Verlag, 2011. (Kapitel 19.10.5, λ-Parser)
- William H. Burge. *Recursive Programming Techniques*. Addison- Wesley, 1975.
- Andy Gill, Simon Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995. `www.haskell.org/happy`
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 8, Functional parsers)
- Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 8(4):437-444, 1998.

Chapter 13.3: Further Reading (7)

- Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
- Daan Leijen. *Parsec, a free Monadic Parser Combinator Library for Haskell*, 2003. `legacy.cs.uu.nl/daan/parsec.html`
- Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. Electronic Notes in Theoretical Computer Science 41(1), 20 pages, 2001.

Chapter 13.3: Further Reading (8)

- Simon Peyton Jones, David Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 10, Code Case Study: Parsing a Binary Data Format)

Chapter 14

Logic Programming Functionally

Logic Programming Functionally

Declarative programming

- Characterizing: Programs are declarative assertions about a problem rather than imperative solution procedures.
- Hence: Emphasizes the 'what,' rather than the 'how.'
- Important styles: Functional and logic programming.

If each of these two styles is appealing for itself

- (features of) functional and logic programming

uniformly combined in just one language should be even more appealing.

Question

- Can and shall (features of) functional and logic programming be uniformly combined?

Yes, they can and should

...a recent article highlights important benefits of combining the paradigm features of functional and logic programming

- Sergio Antoy, Michael Hanus. Functional Logic Programming. Communications of the ACM 53(4):74-85, 2010.

shedding thereby some light on this question.

...part of it is summarized in Chapter 14.1.

Chapter 14.1

Motivation

Chapter 14.1.1

On the Evolution of Programming Languages

The Evolution of Programming Languages (1)

...a continous and ongoing process of hiding the computer hardware and the details of program execution by the stepwise introduction of abstractions.

Assembly languages

- introduce mnemonic instructions and symbolic labels for hiding machine codes and addresses.

FORTRAN

- introduces arrays and expressions in standard mathematical notation for hiding registers.

ALGOL-like languages

- introduce structured statements for hiding gotos and jump labels.

Object-oriented languages

- introduce visibility levels and encapsulation for hiding the representation of data and the management of memory.

Evolution of Programming Languages (2)

Declarative languages, most prominently functional and logic languages

- remove assignment and other control statements for hiding the order of evaluation.
  - A declarative program is a set of logic statements describing properties of the application domain.
  - The execution of a declarative program is the computation of the value(s) of an expression wrt these properties.

This way:

- The programming effort in a declarative language shifts from encoding the steps for computing a result to structuring the application data and the relationships between application components.
- Declarative languages are similar to formal specification languages but executable.

Chapter 14.1.2

Functional vs. Logic Languages

Functional vs. Logic Languages

Functional languages

- are based on the notion of mathematical function.
- programs are sets of functions that operate on data structures and are defined by equations using case distinction and recursion.
- provide efficient, demand-driven evaluation strategies that support infinite structures.

Logic languages

- are based on predicate logic.
- programs are sets of predicates defined by restricted forms of logic formulas, such as Horn clauses (implications).
- provide non-determinism and predicates with multiple input/output modes that offer code reuse.

Functional Logic Languages (1)

...there are many: Curry, TOY, Mercury, Escher, Oz, HAL,...

Some of them in more detail:

- Curry

  Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. Curry: A Truly Functional Logic Language. In Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming, 95-107, 1995.

  See also: Michael Hanus (Ed.). Curry: An Integrated Functional Logic Language (vers. 0.8.2, 2006). `http://www.curry-language.org/`

Functional Logic Languages (2)

- TOY

  Francisco J. López-Fraguas, Jaime Sánchez-Hernández. TOY: A Multi-paradigm Declarative System. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), Springer-V., LNCS 1631, 244-247, 1999.

- Mercury

  Zoltan Somogyi, Fergus Henderson, Thomas Conway. The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language. Journal of Logic Programming 29(1-3):17-64, 1996.

  See also: The Mercury Programming Language
  `http://www.mercurylang.org`

Chapter 14.1.2

A Curry Appetizer

A Curry Appetizer (1)

Two important Curry operators:

- ?, denoting nondeterministic choice.
- =:=, indicating that an equation is to be solved rather than an operation to be defined.

Example: Regular expressions and their semantics

```
data RE a = Lit a
          | Alt (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)

sem :: RE a -> [a]
sem (Lit c)    = [c]
sem (Alt r s)  = sem r ? sem s
sem (Conc r s) = sem r ++ sem s
sem (Star r)   = [] ? sem (Conc r (Star r))
```

A Curry Appetizer (2)

- Evaluating the semantics of the regular expression `abstar`:

```
       non-deterministically
sem abstar ->> ["a","ab","abb"]
 where abstar = Conc (Lit 'a') (Star (Lit 'b'))
```

- Checking whether some word `w` is in the language of a regular expression `re`:

```
sem re =:= w
```

- Checking whether some string `s` contains a word generated by a regular expression `re` (similar to Unix's grep utility):

```
xs ++ sem re ++ ys =:= s
 Note: xs and ys are free!
```

Chapter 14.1.4

Outline

Combining Functional and Logic Programming

...some principal approaches for combining their features:

- Ambitious: Designing a new programming language enjoying features of both programming styles (e.g., Curry, Mercury, etc.).
- Less ambitious: Implementing an interpreter for one style using the other style.
- Even less ambitious: Developing a combinator library allowing us to write logic programs in Haskell.

Here

...we follow the last approach as proposed by Michael Spivey and Silvija Seres in:

- Michael Spivey, Silvija Seres. Combinators for Logic Programming. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 177-199, 2003.

Central are:

- Combinators
- Monads
- Combinator and monadic programming.

Benefits and Limitations

...of this combinator approach compared to approaches striving for fully functional/logic programming languages:

- Less costly

but also

- less expressive and (likely) less performant.

Chapter 14.2

The Combinator Approach

Chapter 14.2.1

Introduction

Three Key Problems

...are to be solved in the course of developing this approach:

Modelling

1. logic programs yielding (possibly) multiple answers
   $\rightsquigarrow$ using the lists of successes technique
2. the evaluation/search strategy inherent to logic programs
   $\rightsquigarrow$ encapsulating the search strategy in 'search monads'
3. logical variables (no distinction between input and output variables)
   $\rightsquigarrow$ realizing unification

Key Problem 1: Multiple Answers

...can easily be handled (re-) using the technique of

- lists of successes (lazy lists) (Philip Wadler, 1985)

Intuitively

- Any function of type `(a -> b)` can be replaced by a function of type `(a -> [b])`.
- Lazy evaluation ensures that the elements of the result list (i.e., the list of successes) are provided as they are found, rather than as a complete list after termination of the computation.

Key Problem 2: Evaluation/Search Strategies

...dealt with investigating an illustrating running example.

This is factoring of natural numbers: Decomposing a positive integer into the set of pairs of its factors, e.g.:

Integer	Factor pairs
24	(1,24), (2,12), (3,8), (4,6), ..., (24,1)

Obviously, this problem (instance) is solved by:

```
factor :: Int -> [(Int,Int)]
factor n = [ (r,s) | r<-[1..n], s<-[1..n], r*s == n]
```

In fact, we get:

```
factor 24 ->>
[(1,24),(2,12),(3,8),(4,6),(6,4),(8,3),(12,2),(24,1)]
```

Note

When implementing the 'obvious' solution we exploit explicit domain knowledge:

- Most importantly the domain fact:
  - $r * s = n \;\Rightarrow\; r \leq n \wedge s \leq n$

  which allows us to restrict our search to a finite space:

$$[1\,..\,24] \times [1\,..\,24]$$

Often, however, such knowledge is not available:

- Generally, the search space cannot be restricted a priori!

In the following, we thus consider the factoring problem as a

- search problem over the infinite 2-dimensional search space:

$$[1\,..\,] \times [1\,..\,]$$

Illustrating the Search Space $[1..] \times [1..]$

	1	2	3	4	5	6	7	8	9	...
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	...
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	...
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	...
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	...
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	...
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	...
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	...
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	...
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	...
...	...	...	...	...	...	...	...	...	...	...

Back to the Running Example

...adapting function `factor` straightforward to the infinite search space `[1..]` $\times$ `[1..]` yields:

$$\texttt{factor :: Int -> [(Int,Int)]}$$
$$\texttt{factor n = [(r,s) | r<-}\underbrace{\texttt{[1..]}}_{\texttt{infinite}}\texttt{, s<-}\underbrace{\texttt{[1..]}}_{\texttt{infinite}}\texttt{, r*s == n]}$$

Applying `factor` to the argument 24 yields:

```
factor 24
->> [(1,24)
```

...followed by an infinite wait.

This is useless and of no practical value!

The Problem: Unfair Depth-first Search

...the two-dimensional space is searched in a depth-first order:

	1	2	3	4	5	6	7	8	9	...
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	...
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	...
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	...
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	...
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	...
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	...
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	...
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	...
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	...
...	...	...	...	...	...	...	...	...	...	...

This search order is unfair: Pairs in rows 2 onwards will never be reached and considered for being a factor pair.

Chapter 14.2.2

Diagonalization

Diagonalization to the Rescue (1)

...searching the infinite number of finite diagonals ensures fairness, i.e., every pair will deterministically be visited after a finite number of steps:

	1	2	3	4	5	6	7	8	9	...
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	...
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	...
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	...
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	...
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	...
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	...
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	...
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	...
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	...
...	...	...	...	...	...	...	...	...	...	...

- Diagonal 1: `[(1,1)]`
- Diagonal 2: `[(1,2),(2,1)]`
- Diagonal 3: `[(1,3),(2,2),(3,1)]`
- Diagonal 4: `[(1,4),(2,3),(3,2),(4,1)]`
- Diagonal 5: `[(1,5),(2,4),(3,3),((4,2),(5,1)]`
- ...

Diagonalization to the Rescue (2)

In fact, on visiting the infinite number of finite diagonals, every pair (i, j) of the infinite 2-dimensional search space $[1..] \times [1..]$ is deterministically reached after a finite number of steps as illustrated below:

	1	2	3	4	5	6	7	...
1	$(1,1)_{\mathbf{1}}$	$(1,2)_{\mathbf{2}}$	$(1,3)_{\mathbf{4}}$	$(1,4)_{\mathbf{7}}$	$(1,5)_{\mathbf{11}}$	$(1,6)_{\mathbf{16}}$	$(1,7)_{\mathbf{22}}$	...
2	$(2,1)_{\mathbf{3}}$	$(2,2)_{\mathbf{5}}$	$(2,3)_{\mathbf{8}}$	$(2,4)_{\mathbf{12}}$	$(2,5)_{\mathbf{17}}$	$(2,6)_{\mathbf{23}}$	$(2,7)_{\mathbf{30}}$	...
3	$(3,1)_{\mathbf{6}}$	$(3,2)_{\mathbf{9}}$	$(3,3)_{\mathbf{13}}$	$(3,4)_{\mathbf{18}}$	$(3,5)_{\mathbf{24}}$	$(3,6)_{\mathbf{31}}$	$(3,7)_{\mathbf{39}}$	...
4	$(4,1)_{\mathbf{10}}$	$(4,2)_{\mathbf{14}}$	$(4,3)_{\mathbf{19}}$	$(4,4)_{\mathbf{25}}$	$(4,5)_{\mathbf{32}}$	$(4,6)_{\mathbf{40}}$	$(4,7)_{\mathbf{49}}$	...
5	$(5,1)_{\mathbf{15}}$	$(5,2)_{\mathbf{20}}$	$(5,3)_{\mathbf{26}}$	$(5,4)_{\mathbf{33}}$	$(5,5)_{\mathbf{41}}$	$(5,6)_{\mathbf{50}}$	$(5,7)_{\mathbf{60}}$	...
6	$(6,1)_{\mathbf{21}}$	$(6,2)_{\mathbf{27}}$	$(6,3)_{\mathbf{34}}$	$(6,4)_{\mathbf{42}}$	$(6,5)_{\mathbf{51}}$	$(6,6)_{\mathbf{61}}$	$(6,7)_{\mathbf{72}}$	...
7	$(7,1)_{\mathbf{28}}$	$(7,2)_{\mathbf{35}}$	$(7,3)_{\mathbf{43}}$	$(7,4)_{\mathbf{52}}$	$(7,5)_{\mathbf{62}}$	$(7,6)_{\mathbf{73}}$	$(7,7)_{\mathbf{85}}$	...
8	$(8,1)_{\mathbf{36}}$	$(8,2)_{\mathbf{44}}$	$(8,3)_{\mathbf{53}}$	$(8,4)_{\mathbf{63}}$	$(8,5)_{\mathbf{74}}$	$(8,6)_{\mathbf{86}}$	$(8,7)_{\mathbf{99}}$	...
9	$(9,1)_{\mathbf{45}}$	$(9,2)_{\mathbf{54}}$	$(9,3)_{\mathbf{64}}$	$(9,4)_{\mathbf{75}}$	$(9,5)_{\mathbf{87}}$	$(9,6)_{\mathbf{100}}$	$(9,7)_{\mathbf{114}}$	...
...	...	...	...	...	...	...	...	...

Homework

The previous figure illustrates that there is a bijective map

$$m : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$$

How can m formally be defined?

Implementing Diagonalization (1)

...function diagprod realizes the diagonalization strategy: It enumerates the cartesian product of its argument lists in a fair order, i.e., every element is enumerated after some finite amount of time:

```
diagprod :: [a] -> [b] -> [(a,b)]
diagprod xs ys =
  [(xs!!i, ys!!(n-i)) | n<-[0..], i<-[0..n]]
                           infinite    finite
```

E.g., applied to the infinite 2-dimensional space $[1..] \times [1..]$, diagprod ejects every pair (x,y) of $[1..] \times [1..]$ in finite time:

```
[(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),
 (3,2),(4,1),(1,5),(2,4),(3,3),(4,2),(5,1),(1,6),
 (2,5),...,(6,1),(1,7),(2,6),...(7,1),...
```

Implementing Diagonalization (2)

```
diagprod :: [a] -> [b] -> [(a,b)]
diagprod xs ys = [(xs!!i, ys!!(n-i)) | n<-[0..], i<-[0..n]]
```

n	i	n-i	(xs!!i, ys!!(n-i))	([1..]!!i, [1..]!!(n-i))	#	Diag. #
0	0	0	(xs!!0,ys!!0)	(1,1)	1	1
1	0	1	(xs!!0,ys!!1)	(1,2)	2	2
1	1	0	(xs!!1,ys!!0)	(2,1)	3	
2	0	2	(xs!!0,ys!!2)	(1,3)	4	3
2	1	1	(xs!!1,ys!!1)	(2,2)	5	
2	2	0	(xs!!2,ys!!0)	(3,1)	6	
3	0	3	(xs!!0,ys!!3)	(1,4)	7	4
3	1	2	(xs!!1,ys!!2)	(2,3)	8	
3	2	1	(xs!!2,ys!!1)	(3,2)	9	
3	3	0	(xs!!3,ys!!0)	(4,1)	10	
4	0	4	(xs!!0,ys!!4)	(1,5)	11	5
4	1	3	(xs!!1,ys!!3)	(2,4)	12	
4	2	2	(xs!!2,ys!!2)	(3,3)	13	
4	3	1	(xs!!3,ys!!1)	(4,2)	14	
4	4	0	(xs!!4,ys!!0)	(5,1)	15	
...	...	...	...	...	...	

Back to the Running Example

...let's adjust `factor` in a way such that it explores the search space of pairs in a fair order using diagonalization:

```
factor :: Int -> [(Int,Int)]
factor n =
```

$$\texttt{[(r,s) | (r,s) <- diagprod } \underbrace{\texttt{[1..]}}_{\texttt{infinite}} \; \overbrace{\texttt{[1..]}}^{\texttt{infinite}}\texttt{, r*s == n]}$$

Applying now `factor` to the argument 24, we obtain:

```
factor 24 ->>
[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)
```

...i.e., we obtain all results, followed by an infinite wait.

Of course, this is not surprising, since the search space is infinite.

Chapter 14.2.3

Diagonalization with Monads

Finite Lists, Infinite Streams, Monads

...in the following we conceptually distinguish between:

- `[a]`: Finite lists.
- `Stream a`: Infinite lists defined as type alias by:

  ```
  type Stream a = [a]
  ```

Note: Distinguishing between `(Stream a)` for infinite lists and `[a]` for finite lists is conceptually and notationally only as is made explicit by defining `(Stream a)` as a type alias of `[a]`.

Like `[]`, `Stream` is a 1-ary type constructor and can thus be made an instance of type class `Monad`:

```
class Monad m where
 return :: a -> m a
 (>>=)  :: m a -> (a -> m b) -> m b
```

Making Stream an Instance of Monad

...since (Stream a) is a type alias of [a], the stream and the list monad coincide; the bind (>>=) and return operation of the stream monad

- (>>=) :: Stream a -> (a -> Stream b) -> Stream b
- return :: a -> Stream a

are thus defined as in Chapter 11.4.2:

```
instance Monad Stream where
 xs >>= f = concat (map f xs)
 return x = [x]        -- yields the singleton list
```

Note: The monad operations (>>) and fail are not relevant in the following, and thus omitted.

Notational Benefit (1)

...the monad operations `return` and `(>>=)` for lists and streams allow us to avoid or replace list comprehension:

E.g., the expression

```
[(x,y) | x <- [1..], y <- [10..]]
```

using list comprehension is equivalent to the expression

```
[1..] >>= (\x -> [10..] >>= (\y -> return (x,y)))
```

using monad operations; this is is made explicit by stepwise unfolding the monadic expression yielding first the equivalent expression:

```
concat (map (\x -> [(x,y) | y <- [10..]])[1..])
```

and second the equivalent expression:

```
concat
 (map (\x -> concat (map (\y -> [(x,y)])[10..]))[1..])
```

Notational Benefit (2)

By exploiting the general rule that

```
do x1 <- e1; x2 <- e2; ... ; xn <- en; e
```

is a shorthand for

```
e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```

...Haskell's do-notation allows an even more compact equivalent representation:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

Note

...exploring the pairs of the search space using the stream monad is not yet fair.

E.g., the expression:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

yields the infinite list (i.e., stream):

```
[(1,10),(1,11),(1,12),(1,13),(1,14),..
```

..the fairness issue is only handled by defining another monad.

Towards a Fair Binding Operation (>>=)

...idea: Embedding diagonalization into (>>=).

To this end, we introduce a new polymorphic type Diag:

```
newtype Diag a = MkDiag (Stream a) deriving Show
```

together with a utility function for stripping off the data constructor MkDiag:

```
unDiag :: Diag a -> a
unDiag (MkDiag xs) = xs
```

Diagonalization with Monads

...making `Diag` an instance of the type constructor class `Monad`:

```
instance Monad Diag where
 return x         = MkDiag [x]
 MkDiag xs >>= f =
  MkDiag (concat (diag (map (unDiag . f) xs)))
```

where `diag` rearranges the values into a fair order:

```
diag :: Stream (Stream a) -> Stream [a]
diag []       = []
diag (xs:xss) =
 lzw (++) [ [x] | x <- xs] ([] : diag xss)
```

Making Diag an Instance of Monad

...using itself the utility function lzw ('like zipWith.'):

```
lzw :: (a -> a -> a) -> Stream a ->
                             Stream a -> Stream a
lzw f [] ys          = ys
lzw f xs []          = xs
lzw f (x:xs) (y:ys) = (f x y) : (lzw f xs ys)
```

Note: lzw equals zipWith except that the non-empty remainder of a non-empty argument list is attached, if one of the argument lists gets empty.

Note

...for monad `Diag` holds:

- `return` yields the singleton list.
- `undiag` strips off the constructor added by the function `f :: a -> Diag b`.
- `diag` arranges the elements of the list into a fair order (and works equally well for finite and infinite lists).

Illustrating

...the idea underlying the map diag:

Transform an infinite list of infinite lists:

```
[[x11,x12,x13,x14,..],[x21,x22,x23,..],[x31,x32,..],..]
```

into an infinite list of finite diagonals:

```
[[x11],[x12,x21],[x13,x22,x31],[x14,x23,x32,..],..]
```

This way, we get:

```
 do x <- MkDiag [1..]; y <- MkDiag [10..]; return (x,y)
  ->> MkDiag [(1,10),(1,11),(2,10),(1,12),(2,11),
             (3,10),(1,13),..
```

which means, we are done:

- The pairs are delivered in a fair order!

Back to the Factoring Problem

...the current status of our approach:

- Generating pairs (in a fair order): Done.
- Selecting the pairs being part of the solution: Still open.

Next, we are going to tackle the selection problem, i.e., filtering out the pairs (r, s) satisfying the equality $r \times s = n$, by:

- Filtering with conditions!

To this end, we introduce a new type constructor class `Bunch`.

Chapter 14.2.4

Filtering with Conditions

The Type Constructor Class Bunch

...is defined by:

```
class Monad m => Bunch m where
 -- Empty result (or no answer)
 zero :: m a
 -- All answers in xm or ym
 alt :: m a -> m a -> m a
-- Answers yielded by ‘auxiliary calculations’
-- (for now, think of wrap in terms of the
-- identity, i.e., wrap = id)
 wrap :: m a -> m a
```

Note: `zero` allows to express that a set of answers is empty; alt allows to join two sets of answers.

Making [] and Diag Instances of Bunch

...making (lazy) lists and Diag instances of Bunch:

```
instance Bunch [] where
 zero       = []
 alt xs ys = xs ++ ys
 wrap xs    = xs

instance Bunch Diag where
 zero    = MkDiag []
 alt (MkDiag xs) (MkDiag ys)       -- shuffle in the
        = MkDiag (shuffle xs ys) -- interest of
 wrap xm = xm                      -- fairness

shuffle :: [a] -> [a] -> [a]
shuffle [] ys      = ys
shuffle (x:xs) ys = x : shuffle ys xs
```

Note: wrap will only be used be used in Chapter 14.2.5 onwards.

Filtering with Conditions using test

Using zero, the function test, which might not look useful at first sight, yields the key for filtering:

```
test :: Bunch m => Bool -> m ()          -- () type idf.
test b = if b then return () else zero -- () value idf.
```

In fact, all do-expressions filter as desired, i.e., the multiples of 3 from the streams [1..] and MkDiag [1..], respectively:

```
do x <- [1..]; () <- test (x `mod` 3 == 0); return x
 ->> [3,6,9,12,15,18,21,24,27,30,33,..
```

```
do x <- [1..]; test (x `mod` 3 == 0); return x
 ->> [3,6,9,12,15,18,21,24,27,30,33,..
```

```
do x <- MkDiag [1..]; test (x `mod` 3 == 0); return x
 ->> MkDiag [3,6,9,12,15,18,21,24,27,30,33,..
```

A note on test

In more detail:

$$\begin{array}{l}
\texttt{do}\ \underbrace{\texttt{x}}_{::\ \texttt{Int}}\ \texttt{<-}\ \underbrace{\texttt{[1..]}}_{::\ \texttt{[] Int}};\\
\quad \underbrace{\texttt{()}}_{::\ \texttt{()}}\ \ \texttt{<-}\ \underbrace{\texttt{test (x `mod` 3 == 0)}}_{\substack{\texttt{[()] :: [] (), if true}\\ \texttt{[] :: [] (), if false}}};\\
\quad \underbrace{\texttt{return x}}_{::\ \texttt{[] Int}}
\end{array}$$

...if `test` evaluates to `true`, it returns the value `()`, and the rest of the program is evaluated. If it evaluates to `false`, it returns `zero`, and the rest of the program is skipped for this value of `x`. This means, `return x` is only reached and evaluated for those values of `x` with ``x `mod` 3`` equals `0`.

Nonetheless

...we are not yet done as the below example shows:

```
do r <- MkDiag [1..]; s <- MkDiag [1..];
   test (r*s==24); return (r,s)
->> MkDiag [(1,24)
```

...followed again by an infinite wait.

Why is that?

The above expression is equivalent to:

```
do x <- MkDiag [1..]
   (do y <- MkDiag [1..]; test(x*y==24);
       return (x,y))
```

Why is that? (1)

...this means the generator for y is merged with the subsequent test to the (sub-) expression:

```
do y <- MkDiag [1..]; test(x*y==24); return (x,y)
```

Intuitively

- This expression yields for a given value of x all values of y with $x * y = 24$.
- For $x = 1$ the answer $(1, 24)$ will be found, in order to then search in vain for further fitting values of y.
- For $x = 5$ we thus would not observe any output, since an infinite search would be initiated for values of y satisfying $5 * y = 24$.

Why is that? (2)

...the deeper reason for this (undesired) behaviour:

The bind operation (>>=) of Diag is not associative, i.e.,

```
xm >>= (\x -> f x >>= g)  =  (xm >>= f) >>= g
```

...does not hold! Or, equivalently expressed using do:

```
do x <- xm; y <- f x; g y
  = xm >>= (\x -> f x >>= (\y -> g y))
  = xm >>= (\x -> f x >>= g)
  = (xm >>= f) >>= g
  = (xm >>= (\x -> f x)) >>= (\y -> g y)
  = do y <- (do x <- xm; f x); g y
```

...does not hold.

Overcoming the Problem

...frankly, `Diag` is not a valid instance of `Monad`, since it fails the monad law of associativity for `(>>=)`. The order of applying generators is thus essential.

For taking this into account, the generators are explicitly pairwise grouped together to ensure they are treated fairly by diagonalization:

```
do (x,y) <- (do u <- MkDiag [1..];
    v <- MkDiag [1..]; return (u,v))
    test (x*y==24); return (x,y)
->> MkDiag [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),
            (1,24),(24,1)
```

...yields now all results, followed, of course, by an infinite wait (due to an infinite search space).

This means, the problem is fixed. We are done.

Note

Getting all results followed by an infinite wait is

- the best we can hope for if the search space is infinite.

Explicit grouping is

- only required because `Diag` is not a valid instance of `Monad` since its bind operation `(>>=)` fails to be associative. If it were, both expressions would be equivalent and explicit grouping unnecessary.

Next, we will strive for

- avoiding/replacing infinite waiting by indicating search progress, i.e., by indicating from time to time that a(nother) result has not (yet) been found.

Chapter 14.2.5

Indicating Search Progress

Indicating Search Progress

...to this end, we introduce a new type `Matrix` together with a cost-guided diagonalization search, a true breadth search.

Intuitively

- Values of type `Matrix`: Infinite lists of finite lists.
- Goal: A program which yields a matrix of answers, where row i contains all answers which can be computed with costs $c(i)$ specific for row i.
- Indicating progress: If the list returned as row k is the empty list, this means 'nothing found,' i.e., the set of solutions which can be computed with costs $c(k)$ is empty.

The Type Matrix

The new type Matrix:

```
newtype Matrix a =
 MkMatrix (Stream [a]) deriving Show
```

...and a utility function for stripping off the data constructor:

```
unMatrix :: Matrix a -> a
unMatrix (MkMatrix xm) = xm
```

Towards `Matrix` an Instance of `Bunch` (1)

...preliminary reasoning about the required operations and their properties:

```
-- Matrix with a single row
return x = MkMatrix [[x]]

-- Matrix without rows
zero = MkMatrix []

-- Concatenating corresponding rows
alt (MkMatrix xm) (MkMatrix ym) =
 MkMatrix (lzw (++) xm ym)

-- Taking care of the cost management!
wrap (MkMatrix xm) = MkMatrix ([]:xm)
```

Towards Matrix an Instance of Bunch (2)

```
{- (>>=) is essentially defined in terms of bindm; it
   handles the data constructor MkMatrix which is not
   done by bindm. -}
(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))

{- bindm is almost the same as (>>=) but without bother-
   ing about MkMatrix; it applies f to all the values
   in xm and collects together the results in a matrix
   according to their total cost: these are the costs
   of the argument of f given by xm plus the cost of
   computing its result. -}
bindm :: Stream[a] -> (a -> Stream[b]) -> Stream [b]
bindm xm f = map concat (diag (map (concatAll . map f) xm))

{- A variant of the concat function using lzw. -}
concatAll :: [Stream [b]] -> Stream [b]
concatAll = foldr (lzw (++)) []
```

Making Matrix an Instance of Bunch

...now we are ready to make Matrix an instance of the type constructor classes Monad and Bunch:

```
instance Monad Matrix where
 return x              = MkMatrix [[x]]
 (MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))

instance Bunch Matrix where
 zero                                = MkMatrix []
 alt( MkMatrix xm) (MkMatrix ym) =
  MkMatrix (lzw (++) xm ym)
 wrap (MkMatrix xm) =    -- ‘wrap xm’ yields a matrix w/
  MkMatrix ([]:xm)       -- the same answers but each
                         -- with a cost one higher than
                         -- its cost in ‘xm’

intMat = MkMatrix [[n] | n <- [1..]]  -- intMat replaces
                                      -- stream [1..]
```

Using intMat and Matrix

...consider the expression:

```
do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
 ->> MkMatrix [[],[],[],[],[],[],[],[],[(4,6),(6,4)],
       [(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],
       [],[],[],[],[],[],[],[(1,24),(24,1)],[],[],[],..
```

Intuitively

- Diagonals 1 to 8: No factor pairs of 24 were found (indicated by []).
- Diagonal 9: The factor pairs (4,6) and (6,4) were found.
- Diagonal 10: The factor pairs (3,8) and (8,3) were found.
- Diagonals 11 to 12: No factor pairs of 24 were found (ind'd by []).
- Diagonal 13: The factor pairs (2,12) and (12,2) were found.
- ...

...if a diagonal d does not contain a valid factor pair, we get `[]`; otherwise we get the list of valid factor pairs located in d.

I.e., we are done: Infinite waiting is replaced by progress indication!

Illustrating the Location

...of the factor pairs of 24 in the diagonals of the search space by **!($\cdot$, $\cdot$)!**:

	1	2	3	4	5	6	7	8	9	...
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	...
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	...
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	**!(3,8)!**	(3,9)	...
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	**!(4,6)!**	(4,7)	(4,8)	(4,9)	...
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	...
6	(6,1)	(6,2)	(6,3)	**!(6,4)!**	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	...
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	...
8	(8,1)	(8,2)	**!(8,3)!**	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	...
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	...
...	...	...	...	...	...	...	...	...	...	

Chapter 14.2.6

Selecting a Search Strategy

An Array of Search Strategies

...is now at our disposal, namely

1. Depth search (`[1..]`)
2. Diagonalization (`MkDiag [[n]| n<-[1..]]`)
3. Breadth search (`MkMatrix [[n]| n<-[1..]]`)

...and we can choose each of them at the very last moment, just by picking the right monad when calling a function:

```
-- Picking the desired search strategy by choos-
-- ing m accordingly at the time of calling factor
factor :: Bunch m => Int -> m (Int, Int)
factor n = do r <- choose [1..]; s <- choose [1..];
              test (r*s==n); return (r,s)

choose :: Bunch m => Stream a -> m a
choose (x:xs) = wrap (return x `alt` choose xs)
```

Picking a Search Strategy at Call Time

...specifying the result type of `factor` when calling it selects the search monad and thus the search strategy applied.

Illustrated in terms of our running example:

```
-- Depth Search: Picking Stream
factor 24 :: Stream (Int,Int)
->> [(1,24)

-- Diagonalization Search: Picking Diag
factor 24 :: Diag (Int, Int)
->> MkDiag [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),
            (1,24),(24,1)
-- Breadth Search w/ Progress Indication: Picking Matrix
factor 24 :: Matrix (Int, Int)
->> MkMatrix [[],[],[],[],[],[],[],[],[(4,6),(6,4)],
     [(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],
     [],[],[],[],[],[],[],[(1,24),(24,1)],[],[],[],..
```

Summarizing our Progress so Far

...recall the 3 key problems we have or had to deal with.

Modelling

1. logic programs yielding (possibly) multiple answers: Done (using lazy lists).
2. the evaluation strategy inherent to logic programs: Done.
   - The search strategy implicit to logic programming languages has been made explicit. The type constructors and type classes of Haskell allow even different search strategies and to pick one conveniently at call time.
3. logical variables (i.e., no distinction between input and output variables): Still open!

Next

...we tackle this third problem, i.e.:

Modelling

- logical variables (i.e., no distinction between input and output variables).

Common for evaluating logic programs

- ...not a pure simplification of an initially completely given expression but a simplification of an expression containing variables, for which appropriate values have to be determined. In the course of the computation, variables can be replaced by other subexpressions containing variables themselves, for which then appropriate values have to be found.

  Fundamental: Substitution, unification.

Chapter 14.2.7

Terms, Substitutions, Unification, and Predicates

Terms (1)

...towards logical variables — we introduce a type for terms:

Terms

```
data Term = Int Int
          | Nil
          | Cons Term Term
          | Var Variable deriving Eq
```

...will describe values of logic variables.

Named variables and generated variables

```
data Variable = Named String
              | Generated Int deriving (Show, Eq)
```

...will be used for formulating queries, respectively, evolve in the course of the computation.

Terms (2)

Utility functions for transforming

- a string into a named variable:

```
var :: String -> Term
var s = Var (Named s)
```

- a list of integers into a term:

```
list :: [Int] -> Term
list xs = foldr Cons Nil (map Int xs)
```

Substitutions (1)

Substitutions

```
newtype Subst = MkSubst [(Var,Term)]
```

...essentially mappings from variables to terms.

Support functions for substitutions:

```
unSubst :: Subst -> [(Var,Term)]
unSubst (MkSubst s) = s

idsubst :: Subst
idsubst = MkSubst []

extend :: Var -> Term -> Subst -> Subst
extend x t (MkSubst s) = MkSubst ((x:t):s)
```

Substitutions (2)

Applying a substitution:

```
apply :: Subst -> Term -> Term
apply s t =              -- Replace each variable
  case deref s t of      -- in t by its image under s
    Cons x xs -> Cons (apply s x) (apply s xs)
    t'        -> t'
```

where

```
deref :: Subst -> Term -> Term
deref s (Var v) =
  case lookup v (unSubst s) of
    Just t    -> deref s t
    Nothing   -> Var v
deref s t = t
```

Term Unification (1)

...unifying terms:

```
unify :: (Term, Term) -> Subst -> Maybe Subst
unify (t,u) s =
  case (deref s t, deref s u) of
    (Nil, Nil) -> Just s
    (Cons x xs, Cons y ys)  ->
                         unify (x,y) s >>= unify (xs, ys)
    (Int n, Int m) | (n==m) -> Just s
    (Var x, Var y) | (x==y) -> Just s
    (Var x, t)              -> if occurs x t s
                               then Nothing
                               else Just (extend x t s)
    (t, Var x)              -> if occurs x t s
                               then Nothing
                               else Just (extend x t s)
    (_,_)                   -> Nothing
```

Term Unification (2)

where

```
occurs :: Variable -> Term -> Subst -> Bool
occurs x t s =
  case deref s t of
    Var y     -> x == y
    Cons y ys -> occurs x y s || occurs x ys s
    _         -> False
```

Predicates: Modelling Logic Programs (1)

...in our scenario `m` is of type `bunch`.

Logic programs are of type:

```
type Pred m = Answer -> m Answer
```

...intuitively, applied to an 'input' answer which provides the information that is already decided about the values of variables, an array of new answers is computed, each of them satisfying the constraints expressed in the program.

Answers are of type:

```
newtype Answer = MkAnswer (Subst,Int)
```

...intuitively, the substitution carries the information about the values of variables; the integer value counts how many variables have been generated so far allowing to generate fresh variables when needed.

Predicates: Modelling Logic Programs (2)

Initial ‘input’ answer:

```
initial :: Answer
initial =  MkAnswer (idsubst, 0)
```

Logical program run: Predicate p as query is applied to the initial ‘input’ answer:

```
run :: Bunch m => Pred m -> m Answer
run p = p initial
```

Example: Choosing Stream for m allows evaluating the predicate append (defined later):

```
run (append (list [1,2],list [3,4],var "z"))
                                   :: Stream Answer
  ->> [{z=[1,2,3,4]}]      -- an appropriate show
                           -- function is assumed
```

Chapter 14.2.8

Combinators for Logic Programs

Combinator (=:=): Equality

...combinator (=:=) ('equality' of terms) allows us to build simple predicates, e.g.:

```
run (var "x" =:= Int 3) :: Stream Answer
  ->> [{x=3}]
```

Implementation of (=:=) by means of unify:

```
(=:=) :: Bunch m => Term -> Term -> Pred m
(t =:= u) (MkAnswer (s,n)) =      -- Pred m = (Answer -> m Answer)
   case unify (t,u) s of
     Just s′ -> return (MkAnswer (s′,n))
     Nothing -> zero
```

Intuitively: If the argument terms t and u can be unified wrt the input answer MkAnswer (s,n), the most general unifier is returned as the output answer; otherwise there is no answer.

Combinator (&&&): Conjunction

...combinator (&&&) allows us to connect predicates conjunctively, e.g.:

```
run (var "x" =:= Int 3 &&& var "y" =:= Int 4)
                                       :: Stream Answer
->> [{x=3,y=4}]
run (var "x" =:= Int 3 &&& var "x" =:= Int 4)
                                       :: Stream Answer
->> []
```

Implementation of (&&&) by means of the bind operation (>>=) of monad bunch:

```
(&&&) :: Bunch m => Pred m -> Pred m -> Pred m
(p &&& q) s = p s >>= q
  -- or equivalently using the do-notation:
  do t <- p s; u <- q t; return u
```

Combinator (|||): Disjunction

...combinator (|||) allows us to connect predicates disjunctively, e.g.:

```
run (var "x" =:= Int 3 ||| var "x" =:= Int 4)
                                    :: Stream Answer
->> [{x=3,x=4}]
```

Implementation of (|||) by means of the alt operation of monad bunch:

```
(|||) :: Bunch m => Pred m -> Pred m -> Pred m
(p ||| q) s = alt (p s) (q s)
```

Assigning Priorities to (=:=), (&&&), (|||)

...is done as follows:

```
infixr 4 =:=
infixr 3 &&&
infixr 2 |||
```

Combinator exists: Existential Quantificat.

...a combinator allowing the introduction of new variables in predicates (exploiting the Int component of answers).

Existential quantification: Introducing local variables in recursive predicates

```
exists :: Bunch m => (Term -> Pred m) -> Pred m
exists p (MkAnswer (s,n)) =
  p (Var (Generated n)) (MkAnswer (s,n+1))
```

Note:

- The term `exists (\x -> ...x...)` has the same meaning as the predicate `...x...` but with `x` denoting a fresh variable which is different from all the other variables used by the program; `n+1` in `MkAnswer (s,n+1)` ensures that never the same variable is introduced by nested calls of `exists`.
- The function `exists` thus takes as its argument a function, which expects a term and produces a predicate; it invents a fresh variable and applies the given function to that variable.

Named vs. Generated Variables

...illustrating the difference:

1)
```
run (var "x" =:= list [1,2,3]
 &&& exists (\t -> var "x" =:= Cons (var "y") t))
                                        :: Stream Answer
 ->> [{x=[1,2,3], y=1}]
```

2)
```
run (var "x" =:= list[1,2,3]
 &&& var "x" =:= Cons (var "y") (var "t"))
                                        :: Stream Answer
 ->> [{t=[2,3], x=[1,2,3], y=1}]
```

Note

- Example 1): The named variable y is set to the head of the list, which is the value of x. The value of the generated variable t is not output.
- Example 2): The same as above but now t denotes a named variable, whose value is output.

Cost Management of Recursive Predicates

...ensuring that in connection with the bunch type `Matrix` the costs per unfolding of the recursive predicate increase by 1 using `wrap`:

```
step :: Bunch m => Pred m -> Pred m
step p s = wrap (p s)
```

Illustrating the usage and effect of `step`:

```
run (var "x" =:= Int 0) :: Matrix Answer
 ->> MkMatrix [[{x=0}]]      -- Without step: Just
                             -- the result.

run (step (var "x" =:= Int 0)) :: Matrix Answer
 ->> MkMatrix [[],[{x=0}]] -- With step: The result
                    -- plus the notification that
               -- there are no answers of cost 0.
```

Chapter 14.2.9

Writing Logic Programs: Two Examples

Writing Logic Programs: Two Examples

We consider two examples:

1. Concatenating lists: The predicate `append`.
2. Testing and constructing 'good' sequences: The predicate `good`.

Example 1: List Concatenation (1)

...implementing a predicate append (a,b,c), where a, b denote lists and c the concatenation of a and b.

The implementation of predicate append:

```
append :: Bunch m => (Term, Term, Term) -> Pred m
append (p,q,r) =
 step (p =:= Nil &&& q =:= r
  ||| exists (\x -> exists (\a -> exists (\b ->
       p =:= Cons x a
       &&& r =:= Cons x b
       &&& append (a,q,b)))))
```

Example 1: List Concatenation (2)

...in more detail:

```
append :: Bunch m => (Term, Term, Term) -> Pred m
append (p,q,r) =
  -- Case 1
  step (p =:= Nil &&& q =:= r
   |||
    -- Case 2
    exists (\x -> exists (\a -> exists (\b ->
     p =:= Cons x a &&& r =:= Cons x b &&& append (a,q,b))))
```

Intuitively

- Case 1: If p is Nil, then r must be the same as q.
- Case 2: If p has the form Cons x a, then r must have the form Cons x b, where b is obtained by recursively concatenating a with the unchanged q.
- Termination: Is ensured since the third argument is getting smaller in each recursive call of append.

Example 1: List Concatenation (3)

...as common for logic programs, there is no difference between input and output variables. Hence, multiple usages of append are possible, e.g.:

a) Using append for concatenating two lists:

```
run (append (list [1,2], list [3,4], var "z"))
                                        :: Stream Answer
  ->> [{z=[1,2,3,4]}]
      -- An appropriate implementation of show
      -- generating the above output is assumed.

      -- More closely related to the internal structure
      -- of the value of z would be an output like:
        ->> Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

Example 1: List Concatenation (4)

Using append for computing the set of lists which equal a given list

b) ...when concatenated:

```
run (append (var "x", var "y", list [1,2,3]))
                                        :: Stream Answer
->> [{x = Nil, y = [1,2,3]},
     {x = [1], y = [2,3]},
     {x = [1,2], y = [3]},
     {x = [1,2,3], y = Nil}]
```

c) ...when concatenated with another given list:

```
run (append (var "x", list [2,3], list [1,2,3]))
                                        :: Stream Answer
->> [{x = [1]}]
```

Example 2: 'Good' Sequences (1)

...implementing a predicate good allowing to

- generate sequences of 0s and 1s, which are considered 'good.'
- check, if a sequence of 0s and 1s is 'good.'

We define:

1. The sequence `[0]` is good.
2. If the sequences `s1` and `s2` are good, then also the sequence `[1] ++ s1 ++ s2`.
3. There is no other good sequence except of those formed in accordance to the above two rules.

Example 2: 'Good' Sequences (2)

Examples:

- 'Good' sequences

  [0]

  [1]++[0]++[0] = [100]

  [1]++[0]++[100] = [10100]

  [1]++[100]++[0] = [11000]

  [1]++[100]++[10100] = [110010100]

  ...

- 'Bad' sequences

  [1], [11], [110], [000], [010100], [1010101],...

Example 2: 'Good' Sequences (3)

Lemma 14.2.9.1 (Properties of 'Good' Sequences)

If a sequence s is good, then

1. the length of s is odd
2. $s = [0]$ or there is a sequence t with $s = [1] {+\!+} t {+\!+} [00]$

Note: The converse implication of Lemma 14.2.8.1(2) does not hold: the sequence $[11100] = [1] {+\!+} [11] {+\!+} [00]$, e.g., is bad.

Example 2: 'Good' Sequences (4)

The implementation of predicate good:

```
good ::  Bunch m => Term -> Pred m
good (s) =
 step (s =:= Cons (Int 0) Nil
  ||| exists (\t -> exists (\q -> exists (\r ->
      s =:= Cons (Int 1) t
      &&& append (q,r,t)
      &&& good (q)
      &&& good (r)))))
```

Example 2: 'Good' Sequences (5)

...in more detail:

```
good ::  Bunch m => Term -> Pred m
good (s) =
 step (
 -- Case 1
  s =:= Cons (Int 0) Nil
   |||
    -- Case 2
    exist (\t -> exists (\q -> exists (\r ->
     s =:= Cons (Int 1) t
     &&& append (q,r,t) &&& good (q) &&& good (r)))))
```

Intuitively

- Case 1: Checks if `s` is `[0]`.
- Case 2: If `s` has the form `[1]++t` for some sequence `t`, all ways are checked of splitting `t` into two sequences `q` and `r` with `q++r==t` and `q` and `r` are good sequences themselves.
- Termination: Is ensured, since `t` gets smaller in every recursive call and the number of its splittings is finite.

Example 2: 'Good' Sequences (6)

Using predicate good.

1) Checking if a sequence is good using Stream:

```
run (good (list [1,0,1,1,0,0,1,0,0]))
                                        :: Stream Answer
  ->> [{}] -- Returning the empty set as answer,
           -- if the argument list is good.

run (good (list [1,0,1,1,0,0,1,0,1]))
                                        :: Stream Answer
  ->> []   -- Returning no answer, if the argument
           -- list is bad.
```

Note: The "empty answer" and the "no answer" correspond to the answers "yes" and "no" of a Prolog system.

Example 2: 'Good' Sequences (7)

2a) Constructing good sequences using Stream:

```
run (good (var "s")) :: Stream Answer
 ->> [{s=[0]},
      {s=[1,0,0]},
      {s=[1,0,1,0,0]},
      {s=[1,0,1,0,1,0,0]},
      {s=[1,0,1,0,1,0,1,0,0]},..
```

...some answers will not be generated, since the depth search induced by Stream is not fair. The computation is thus likely to get stuck at some point.

Example 2: 'Good' Sequences (8)

2b) Constructing good sequences using Diag:

```
run (good (var "s")) :: Diag Answer
 ->> Diag [{s=[0]},
           {s=[1,0,0]},
           {s=[1,0,1,0,0]},
           {s=[1,0,1,0,1,0,0]},
           {s=[1,1,0,0,0]},
           {s=[1,0,1,0,1,0,1,0,0]},
           {s=[1,1,0,0,1,0,0]},
           {s=[1,0,1,1,0,0,0]},
           {s=[1,1,0,0,1,0,1,0,0]},..
```

...eventually all answers will be generated, since the diagonalization search induced by Diag is fair. However, the output order can hardly be predicted due to the interaction of diagonalization and shuffling.

Example 2: 'Good' Sequences (9)

2c) Constructing good sequences using Matrix:

```
run (good (var "s")) :: Matrix Answer
 ->> MkMatrix [[],
       [{s=[0]}],[],[],[],
       [{s=[1,0,0]}],[],[],[],
       [{s=[1,0,1,0,0]}],[],
       [{s=[1,1,0,0,0]}],[],
       [{s=[1,0,1,0,1,0,0]}],[],
       [{s=[1,0,1,1,0,0,0]},{s=[1,1,0,0,1,0,0]}],[],
       ..
```

...using the cost-guided 'true' breadth search induced by Matrix, the output order of results seems more 'predictable' than for the search induced by Diag. Additionally, we get 'progress notifications.'

Remarks on Missing Code / Homework

Note, code for

- pretty printing terms and answers
- making the types `Term`, `Subst`, and `Answer` instances of the type class `Show`

is missing and must be provided by a user of the approach.

Chapter 14.3

Summary

Summing up

Current functional logic languages aim at balancing

- generality (in terms of paradigm integration).
- efficiency of implementations.

Functional logic programming offers

- support of specification, prototyping, and application programming within a single language.
- terse, yet clear, support for rapid development by avoiding some tedious tasks, and allowance of incremental refinements to improve efficiency.

Overall: Functional logic programming is

- an emerging paradigm with appealing features.

Chapter 14.4

References, Further Reading

Chapter 14: Further Reading (1)

- Hassan Ait-Kaci, Roger Nasr. *Integrating Logic and Functional Programming*. Lisp and Symbolic Computation 2(1):51-89, 1989.
- Sergio Antoy, Michael Hanus. *Compiling Multi-Paradigm Declarative Languages into Prolog*. In Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2000), Springer-V., LNCS 1794, 171-185, 2000.
- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
- Sergio Antoy, Michael Hanus. *New Functional Logic Design Patterns*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 19-34, 2011.

Chapter 14: Further Reading (2)

- Bernd Braßel, Michael Hanus, Björn Peemöller, Fabian Reck. *KiCS2: A New Compiler from Curry to Haskell*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 1-18, 2011.
- Norbert Eisinger, Tim Geisler, Sven Panne. *Logic Implemented Functionally*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-V., LNCS 1292, 351-368, 1997.

Chapter 14: Further Reading (3)

- Michael Hanus (Ed.). *Curry: An Integrated Functional Logic Language*. Vers. 0.8.2, 2006.
  www.curry-language.org/
  Vers. 0.8.3, September 11, 2012:
  http://www.informatik.uni-kiel.de/∼curry/report.html
- Michael Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice*. Journal of Functional Programming 19&20:583-628, 1994.
- Michael Hanus. *Multi-paradigm Declarative Languages*. In Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Springer-V., LNCS 4670, 45-75, 2007.

Chapter 14: Further Reading (4)

- Michael Hanus. *Functional Logic Programming: From Theory to Curry*. In *Programming Logics – Essays in Memory of Harald Ganzinger*. Springer-V., LNCS 7797, 123-168, 2013.
- Michael Hanus, Sergio Antoy, Bernd Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner. *PAKCS: The Portland Aachen Kiel Curry System*. 2013. Available at `www.informatik.uni-kiel.de/~pakcs`
- Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. *Curry: A Truly Functional Logic Language*. In Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming, 95-107, 1995.

Chapter 14: Further Reading (5)

- Joxan Jaffar, Jean-Louis Lassez. *Constraint Logic Programming*. In Conference Record of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'87), 111-119, 1987.
- John W. Lloyd. *Programming in an Integrated Functional and Logic Language*. Journal of Functional and Logic Programming 1999(3), 49 pages, MIT Press, 1999.
- Francisco J. López-Fraguas, Jaime Sánchez-Hernández. *TOY: A Multi-paradigm Declarative System*. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), Springer-V., LNCS 1631, 244-247, 1999.

Chapter 14: Further Reading (6)

- Kimbal Marriott, Peter J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 22, Integration von Konzepten anderer Programmiersprachen)
- Uday S. Reddy. *Narrowing as the Operational Semantics of Functional Languages*. In Proceedings of the IEEE International Symposium on Logic Programming, 138-151, 1985.
- Tom Schrijvers, Peter J. Stuckey, Philip Wadler. *Monadic Constraint Programming*. Journal of Functional Programming 19(6):663-697, 2009.

Chapter 14: Further Reading (7)

- Zoltan Somogyi, Fergus Henderson, Thomas Conway. *The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language*. Journal of Logic Programming 29(1-3):17-64, 1996.

- Zoltan Somogyi, Fergus J. Henderson, Thomas C. Conway. *Mercury: An Efficient Purely Declarative Logic Programming Language*. In Proceedings of the 18th Australasian Computer Science Conference, 499-512, 1995.

- Silvija Seres, Michael Spivey. *Embedding Prolog in Haskell*. In Proceedings of the 1999 Haskell Workshop (Haskell'99), Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, 25-38, 1999.

Chapter 14: Further Reading (8)

- Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 177-199, 2003.
- Philip Wadler. *How to Replace Failure with a List of Successes*. In Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 113-128, 1985.

Chapter 15

Pretty Printing

Chapter 15.1

Motivation

Pretty Printing

...is about

- 'beautifully' printing values of tree-like structures as plain text.

A pretty printer is a

- tool (often a library of routines) designed for converting a tree value into plain text

such that the

- tree structure is preserved and reflected by indentation while utilizing a minimum number of lines to display the tree value.

Pretty printing can thus be considered

- dual to parsing.

Pretty Printing

...is just as parsing often used for demonstrating the power and elegance of functional programming, where not just the

- printed result of a pretty printer shall be 'pretty'
- but also the pretty-printer ifself including that its code is short and fast, and its operators enjoy properties which are appealing from a mathematical point of view.

Overall, a 'good' pretty printer must properly balance:

- Ease of use
- Flexibility of layout
- 'Beauty' of output

...while being ifself 'pretty.'

The Prettier Printer

...presented in this chapter has been proposed by Philip Wadler in:

- Philip Wadler. A Prettier Printer. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

which has been designed to improve (cf. Chapter 15.5) on a pretty printer proposed by John Hughes which is widely recognized as a standard:

- John Hughes. The Design of a Pretty-Printer Library. In Johan Jeuring, Erik Meijer (Eds.), Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques. Springer-V., LNCS 925, 53-96, 1995.

Outline and Assumptions

...the implementation of the simple pretty printer and the prettier printer of Philip Wadler assumes some implementation of a type of documents `Doc`.

The

- simple pretty printer (cf. Chapter 15.2)
  - implements `Doc` as strings.
  - supports for every document only one possible layout, in particular, no attempt is made to compress structure onto a single line.
- prettier printer (cf. Chapter 15.3)
  - implements `Doc` in terms of suitbable algebraic sum data types.
  - allows multiple layouts of a document and to pick a best one out of them for printing a document.

Chapter 15.2

The Simple Pretty Printer

Chapter 15.2.1

Basic Document Operators

The Simple Pretty Printer

...(as well as the prettier printer later on) relies on six basic document operators:

Associative operator for concatenating documents:

```
(<>) :: Doc -> Doc -> Doc
```

The empty document being a right and left unit for (<>):

```
nil :: Doc
```

Converting a string into a document (arguments of function text shall not contain newline characters):

```
text :: String -> Doc
```

The document representing a line break:

```
line :: Doc
```

Adding indentation to a document:

```
nest :: Int -> Doc -> Doc
```

Layouting a document as a string:

```
layout :: Doc -> String
```

String Documents

...choosing for the simple pretty printer strings for implementing documents, i.e.:

- `type Doc = String`

the implementation of the basic operators boils down to:

- `(<>)`: String concatenation `++`.
- `nil`: The empty string `[]`.
- `text`: The identity on strings.
- `line`: The string formed by the newline character `'\n'`.
- `nest i`: indentation, adding `i` spaces (only used after line breaks by means of `line`).
- `layout`: The identity on strings.

Note

...the coupling of `line` and `nest` is an essential difference to the pretty printer of John Hughes, where insertion of spaces is also allowed in front of strings.

This difference is key for succeeding with only one concatenation operator for documents instead of the two in the pretty printer of John Hughes (cf. Chapter 15.5).

Chapter 15.2.2

Normal Forms of String Documents

String Documents

...can always be reduced to a normal form representation alternating applications of function

- text with line breaks nested to a given indentation:

  `text s_0 <> nest i_1 line <> text s_1 <> ...`
  `<> nest i_k line <> text s_k`

where every

- s_j is a string (possibly empty).
- i_j is a natural number (possibly zero).

Example: Normal Form Representation

The document (i.e., a Doc-value):

```
text "bbbbb" <> text "[" <>
nest 2 (
     line <> text "ccc" <> text "," <>
     line <> text "dd"
) <>
line <> text "]" :: Doc
```

which prints as:

```
bbbbb[
  ccc,
  dd
]
```

has the normal form (representation):

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"  :: Doc
```

Normal Form Representations

...of string documents exist because of a variety of laws the basic operators of the simple pretty printer enjoy. In particular:

Lemma 15.2.2.1 (Associativity of Doc. Concatenat.)

`(<>)` is associative with unit `nil`.

...as well as the collection of basic operator laws compiled in Lemma 15.2.2.2.

Basic Operators Laws

Lemma 15.2.2.2 (Basic Operator Laws)

1. Operator text is a homomorphism from string to document concatenation:

```
text (s ++ t)    = text s <> text t
text ""          = nil
```

2. Opr. nest is a homomorph. from addition to composition:

```
nest (i+j) x     = nest i (nest j x)
nest 0 x         = x
```

3. Opr. nest distributes through document concatenation:

```
nest i (x <> y) = nest i x <> nest i y
nest i nil      = nil
```

4. Nesting is absorbed by text (differently to the pretty printer of Hughes):

```
nest i (text s) = text s
```

Note

...the laws compiled in Lemma 15.2.2.1 and 15.2.2.2

- come, except of the last one, in pairs with a corresponding law for the unit of the respective operator.
- are sufficient to ensure that every document can be transformed into normal form, where the
  - laws of part 1) and 2) are applied from left to right.
  - last of part 3) and 4) are applied from right to left.

Laws

...relating string documents with their layouts:

Lemma 15.2.2.3 (Layout Operator Laws)

1. Operator layout is a homomorphism from document to string concatenation:

```
layout (x <> y)      = layout x ++ layout y
layout nil           = ""
```

2. Operator layout is the inverse of function text:

```
layout (text s)      = s
```

3. The result of layout applied to a nested line is a newline followed by one space for each level of indentation:

```
layout (nest i line) = '\n' : copy i ' '
```

Chapter 15.2.3

Printing Trees

Using the Simple Pretty Printer

...for prettily printing values of the data type Tree defined by:

```
data Tree = Node String [Tree]
```

For illustration, consider Tree-value t:

```
t = Node "aaa"
      [Node "bbbbb" [Node "ccc" [],Node "dd" []],
       Node "eee" [],
       Node "ffff"
         [Node "gg" [],Node "hhh" [],Node "ii" []]]
```

Two different Layouts of *t* as Strings

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

where `t = Node "aaa"`

```
          [Node "bbbbb" [Node "ccc" [],Node "dd" []],
           Node "eee" [],
           Node "ffff"
            [Node "gg" [],Node "hhh" [],Node "ii" []]]
```

The Layout Strategies

...used for layouting and printing tree `t`:

- Left: Tree siblings start on a new line, properly indented.
- Right: Every subtree starts on a new line, properly indented by two spaces.

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

Implementing the 'Left' Layout Strategy

...by means of a utility function showTree converting a tree into a string document according to the 'left' layout strategy:

```
type Doc  = String
data Tree = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) =
 text s <> nest (length s) (showBracket ts)

showBracket :: [Tree] -> Doc
showBracket [] = nil
showBracket ts =
 text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees :: [Tree] -> Doc
showTrees [t]    = showTree t
showTrees (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

Implementing the ‘Right’ Layout Strategy

...by means of a utility function showTree′ converting a tree into a string document according to the ‘right’ layout strategy:

```
type Doc  = String
data Tree = Node String [Tree]

showTree' :: Tree -> Doc
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' :: [Tree] -> Doc
showBracket' [] = nil
showBracket' ts =
 text "[" <> nest 2 (line <> showTrees' ts) <> line
                                              <> text "]"

showTrees' :: [Tree] -> Doc
showTrees' [t]    = showTree t
showTrees' (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

Chapter 15.3

The Prettier Printer

Chapter 15.3.1

Algebraic Documents

Algebraic Documents

...for the prettier printer we consider a document a

- concatenation of items, where each item is a text or a line break indented a given amount.

Documents are thus implemented as an algebraic sum data type:

```
data Doc = Nil
         | String `Text` Doc
         | Int `Line` Doc
```

Note, the data constructors `Nil`, `Text`, and `Line` of `Doc` relate to the basic document operators `nil`, `text`, and `line` of the simple pretty printer as follows:

```
(1)  Nil          ≙  nil
(2)  s `Text` x   ≙  text s <> x
(3)  i `Line` x   ≙  nest i line <> x
```

Example: String vs. Algebraic Document Rep.

...the normal form representation of the string document considered in Chapter 15.2.2:

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

...is represented by the algebraic Doc-value:

```
"bbbbb[" `Text` (
2 `Line` ("ccc," `Text` (
2 `Line` ("dd," `Text` (
0 `Line` ("]," `Text` Nil))))))
```

Chapter 15.3.2

Implementing Document Operators on Algebraic Documents

Implementations

...of the basic document operators on algebraic documents can easily be derived from 'equations' (1) - (3) of Chapter 15.3.1:

```
nil                     =  Nil
text s                  =  s `Text` Nil
line                    =  0 `Line` Nil

Nil <> y                =  y
(s `Text` x) <> y       =  s `Text` (x <> y)
(i `Line` x) <> y       =  i `Line` (x <> y)

nest i Nil              =  Nil
nest i (s `Text` x)     =  s `Text` nest i x
nest i (j `Line` x)     =  (i+j) `Line` nest i x

layout Nil              =  ""
layout (s `Text` x)     =  s ++ layout x
layout (i `Line` x)     =  `\n` : copy i ` ` ++ layout x
```

Justification

...for the derived definitions can be given using equational reasoning, e.g.:

Proposition 15.3.2.1

```
(s `Text` x) <> y = s `Text` (x <> y)
```

Proof by equational reasoning.

```
    (s `Text` x) <> y
=    { Definition of Text, equ. (2) }
    (text s <> x) <> y
=    { Associativity of <> }
     text s <> (x <> y)
=    { Definition of Text, equ. (2) }
     s `Text` (x <> y)
```

□

...similarly, correctness of the other equations from the previous slide can be shown.

Chapter 15.3.3

Multiple Layouts of Algebraic Documents

Single vs. Multiple Layouts of Documents

...so far, a document d could essentially be considered equivalent to a

- single string defining a unique single layout for d.

Next, a document shall be considered equivalent to a

- set of strings, each of them defining a layout for d, together thus multiple layouts.

To achieve this, only one new document operator must be added:

```
group :: Doc -> Doc
group x = flatten x <|> x
```

with `flatten` and `(<|>)` to be defined soon.

The Meaning of group

...applied to a document representing a set of layouts, group

- returns the set with one new element added representing the layout, in which everything is compressed on one line.

This is achieved by

- replacing each newline (and the corresponding indentation) with text consisting of a single space.

Note: Variants where

- each newline carries with it the alternate text it should be replaced with

are possible, e.g. some newlines might be replaced by the empty text, others by a single space (but are not considered here).

The relative 'Beauty' of a Layout

...depends much on the preferred maximum line width considered eligible for a layout.

Therefore, the document operator `layout` used so far is replaced by a new operator `pretty`:

```
pretty :: Int -> Doc -> String
```

which picks the 'prettiest' among a set of layouts depending on the `Int`-value of the preferred maximum line width argument.

Example

...replacing `showTree` of the 'left' layout strategy for trees of Chapter 15.2.3:

```
data Tree = Node String [Tree]
```

```
showTree :: Tree -> Doc
showTree (Node s ts) =
  text s <> nest (length s) (showBracket ts)
```

by a refined version with an additional call of `group`:

```
showTree (Node s ts) =
  group (text s <> nest (length s) (showBracket ts))
```

will ensure that

- trees are fit onto one line where possible ($\leq$ *max* width).
- sufficiently many line breaks are inserted in order to avoid exceeding the preferred maximum line width.

Example (cont'd)

...calling, e.g., `pretty 30` will (when completely specified!) yield the output:

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

Defining the new Operators (<|>), `flatten`

...for completing the implementation of the operators `group` and `pretty`.

Union operator, forming the union of two sets of layouts:

```
(<|>) :: Doc -> Doc -> Doc
```

Flattening operator, replacing each line break (and its associated indentation) by a single space:

```
flatten :: Doc -> Doc
```

Note: The operators `<|>` and `flatten` will not directly exposed to the user but only via `group` and the operators `fillwords` and `fill` defined in Chapter 15.3.6.

Required Invariant for `(<|>)`

...assuming that a document always represents a non-empty set of layouts, which all flatten to the same layout, the following invariant for the union operator `(<|>)` is required:

- Invariant: In `(x <|> y)` all layouts of `x` and `y` flatten to the same layout.

...this invariant must be ensured when creating a union `(<|>)`.

Distribution Laws

...required for the implementations of `(<|>)` and `flatten`.

Each operator on simple documents extends pointwise through union:

Distributive Laws for `(<|>)`

1. `(x <|> y) <> z     = (x <> z) <|> (y <> z)`
2. `x <> (y <|> z)     = (x <> y) <|> (x <> z)`
3. `nest i (x <|> y) = nest i x <|> nest i y`

Since flattening gives the same result for each element of a set, the distribution law for `flatten` is simpler:

Distributive Law for `flatten`

```
flatten (x <|> y) = flatten x
```

Interaction Laws

...required for the implementation of `flatten`.

Concerning the interaction of `flatten` with other document operators:

Interaction Laws for `flatten`

1. `flatten (x <> y)     = flatten x <> flatten y`
2. `flatten nil          = nil`
3. `flatten (text s)     = text s`
4. `flatten line         = text " "`
5. `flatten (nest i x)   = flatten x`

Note, laws (4) and (5) are the most interesting ones:

- (4): linebreaks are replaced by a single space.
- (5): indentations are removed.

Recalling the Implementation

...of `group` in terms of `flatten` and `(<|>)`:

```
group :: Doc -> Doc
group x = flatten x <|> x
```

Recall, too:

- Documents always represent a non-empty set of layouts whose elements all flatten to the same layout.
- `group` adds the flattened layout to a set of layouts.

Chapter 15.3.4

Normal Forms of Algebraic Documents

Normal Form Representations

...due to the laws for flattening (`flatten`) and union (`(<|>)`) every document can be reduced to a representation in normal form of the form:

`x_1 <|> ... <|> x_n`

where every `x_j` is in the normal form of simple documents (cf. Chapter 15.2.2).

Picking a 'prettiest' Layout

...out of a set of layouts is done by means of an ordering relation on lines depending on the preferred maximum line width, and extended lexically to an ordering between documents.

Out of two lines

- which both do not exceed the maximum width, pick the longer one.
- of which at least one exceeds the maximum width, pick the shorter one.

Note: These rules reqire to pick sometimes a layout where some lines exceed the limit. This is an important difference to the approach of John Hughes, done only, however, if unavoidable.

Adapting the Algebraic Definition of Doc

...the algebraic definition of Doc of Chapter 15.3.1 is extended by a new data vconstructor Union representing the union of two documents:

```
data Doc = Nil
         | String `Text` Doc
         | Int `Line` Doc
         | Doc `Union` Doc    -- Union, the new
                              -- data constructor!
```

Note, these data value constructors relate to the basic document operators as follows:

(1)	`Nil`	$\widehat{=}$	`nil`
(2)	``s `Text` x``	$\widehat{=}$	`text s <> x`
(3)	``i `Line` x``	$\widehat{=}$	`nest i line <> x`
(4)	``x `Union` y``	$\widehat{=}$	`x <|> y`

Required Invariants for `Union`

...assuming again that a document always represents a non-empty set of layouts flattening all to the same layout, two invariants are required for `Union`:

- Invariant 1: In `(x ‘Union‘ y)` all layouts of `x` and `y` flatten to the same layout.
- Invariant 2: Every first line of a document in `x` is at least as long as every first line of a document in `y`.

...these invariants must be ensured when creating a `Union`.

Performance

...of pretty printing is improved by applying the distributive law for `Union` giving

```
(s `Text` (x `Union` y))
```

preference to the equivalent

```
((s `Text` x) `Union` (s `Text` y))
```

Illustrating the Performance Impact (1)

...of distributivity considering the document:

```
group(
   group(
        group(
           group(text "hello" <> line <> text "a")
        <> line <> text "b")
    <> line <> text "c")
<> line <> text "d")
```

...and its possible layouts:

```
hello a b c d    hello a b c    hello a b    hello a    hello
                 d              c            b          a
                                d            c          b
                                             d          c
                                                        d
```

Illustrating the Performance Impact (2)

...printing the previous document within a maximum line width of 5, its

- right-most layout must be picked

...ideally, while the other ones are eliminated in one fell swoop.

Intuitively, this is achieved by picking a representation, which brings to the front any common string, e.g.:

```
"hello" `Text` ((" ") `Text` x) `Union` (0 `Line` y))
```

for suitable documents `x` and `y`, where `"hello"` has been factored out of all the layouts in `x` and `y`, and `" "` of all the layouts in `x`.

Since `"hello"` followed by `" "` is of length 6 exceeding the limit 5, the right operand of `Union` can immediately be chosen without further examination of `x`, as desired.

Fixing the Performance Issue

...to realize this, `(<>)` and `nest` must be extended to specify how they interact with `Union`:

```
(x `Union` y) <> z    = (x <> z) `Union` (y <> z)   (1)
nest k (x `Union` y) = nest k x `Union` nest k y   (2)
```

while the definitions of `nil`, `text`, `line`, `(<>)`, and `nest` remain unchanged.

Note, (1) and (2) follow from the distributive laws. In particular, they preserve Invariant 2 required by `Union`.

Algebraic Definitions

...of group and flatten are then easily derived:

```
group Nil            = Nil
group (i `Line` x)   = (" " `Text` flatten x)
                          `Union` (i `Line` x)
group (s `Text` x)   = s `Text` group x
group (x `Union` y)  = group x `Union` y

flatten Nil           = Nil
flatten (i `Line` x)  = " " `Text` flatten x
flatten (s `Text` x)  = s `Text` flatten x
flatten (x `Union` y) = flatten x
```

Justification (1)

...for the derived definitions can be given using equational reasoning, e.g.:

Proposition 15.3.4.1

```
group (i `Line` x) =
        (" " `Text` flatten x) `Union` (i `Line` x)
```

Proof by equational reasoning.

```
   group (i `Line` x)
=  {Definition of Line, equ. (3)}
   group (nest i line <> x)
=  {Definition of group}
   flatten (nest i line <> x) <|> (nest i line s <> x)
=  {Definition of flatten}
   (text " " <> flatten x) <|> (nest i line <> x)
=  {Definition of Text, Union, Line, equ. (2), (4), (3)}
   (" " `Text` flatten x) `Union` (i `Line` x)
```

□

Justification (2)

Proposition 15.2.4.5

```
group (s `Text` x) = s `Text` group x
```

Proof by equational reasoning.

```
   group (s `Text` x)
=  {Definition of Text, equ. (2)}
   group (text s <> x)
=  {Definition of group}
   flatten (text s <> x) <|> (text s <> x)
=  {Definition of flatten}
   (text s <> flatten x) <|> (text s <> x)
=  {(<>) distributes through (<|>)}
   text s <> (flatten x <|> x)
=  {Definition of group}
   text s <> group x
=  {Definition of Text, equ. (2)}
   s `Text` group x
```

□

Picking the ‘best’ Layout (1)

...among a set of layouts using functions best and better:

```
best w k Nil              = Nil
best w k (i `Line` x)   = i `Line` best w i x
best w k (s `Text` x)
         = s `Text` best w (k + length s) x
best w k (x `Union` y)
         = better w k (best w k x) (best w k y)

better w k x y
         = if fits (w-k) x then x else y
```

Note:

- best: Converts a ‘union’-afflicted document into a ‘union’-free document.
- Argument w: Maximum line width.
- Argument k: Already consumed letters (including indentation) on current line.

Picking the 'best' Layout (2)

Check, if the first document line stays within the maximum line length `w`:

```
fits w x | w<0       = False -- cannot fit
fits w Nil           = True  -- fits trivially
fits w (s `Text` x)
  = fits (w - length s) x   -- fits if x fits into
                            -- the remaining space
                            -- after placing s
fits w (i `Line` x) = True  -- yes, it fits
```

Last but not least, the output routine: Pick the best layout and convert it to a string:

```
pretty w x = layout (best w 0 x)
```

Chapter 15.3.5

Improving Performance

Intuitively

...pretty printing a document should be doable in time $\mathcal{O}(s)$, where s is the size of the document, i.e., a count of

- the number of `(<>)`, `nil`, `text`, `nest`, and `group` operations
- plus the length of all string arguments to `text`.

and in space proportional to $\mathcal{O}(w \ max \ d)$, where

- w is the width available for printing
- d is the depth of the document, the depth of calls to `nest` or `group`.

Sources of Inefficiency

...of the prettier printer implementation so far:

1. Document concatenation might pile up to the left:

   ```
   (...((text s_0 <> text s_1) <> ...) <> text s_n
   ```

   ...assuming each string has length one, this may require time $\mathcal{O}(n^2)$ to process (instead of $\mathcal{O}(n)$ as hoped for).

2. Nesting of documents adds a layer of processing to increment the indentation of the inner document:

   ```
   nest i_o (text s_0 <> nest i_1 (text s_1 <>
                           ... <> nest i_n (text s_n)...))
   ```

   ...even if we assume document concatenation associates to the right.

   ...assuming again each string has length one, this may also require time $\mathcal{O}(n^2)$ to process (instead of $\mathcal{O}(n)$ as hoped for).

Performance Fixes

...for inefficiency source 1):

- Adding an explicit representation for concatenation, and generalizing each operation to act on a list of concatenated documents.

...for inefficiency source 2):

- Adding an explicit representation for nesting, and maintaining a current indentation that is incremented as nesting operators are processed.

Combining both fixes suggests

- generalizing each operation to work on a list of indentation-document pairs.

Implementing the Fixes

...by switching to a new representation for documents such that there is one data constructor for every operator building a document:

```
data DOC = NIL
         | DOC :<> DOC
         | NEST Int DOC
         | TEXT String
         | LINE
         | DOC :<|> DOC
```

Note: To avoid name clashes with the previous definitions, capital letters are used.

Implementing the Document Operators

...building a document of the new algebraic type is straightforward:

```
nil        = NIL
x <> y     = x :<> y
nest i x = NEST i x
text s     = TEXT s
line       = LINE
```

As before, also the invariants on the equality of flattened layouts and on the relative lengths of first lines are required:

- In `(x :<|> y)` all layouts in `x` and `y` flatten to the same layout.
- No first line in `x` is shorter than any first line in `y`.

Implementing `group` and `flatten`

...for the new algebraic type is straightforward, too:

```
group x             = flatten x :<|> x

flatten NIL         = NIL
flatten (x :<> y)   = flatten x:<> flatten y
flatten (NEST i x) = NEST i (flatten x)
flatten (TEXT s)    = TEXT s
flatten LINE        = TEXT " "
flatten (x :<|> y) = flatten x
```

...the definitions follow immediately from the equations given before.

The Representation Function rep

...maps a list of indentation-document pairs into the corresponding document:

```
rep z = fold (<>) nil [nest i x | (i,x) <- z]
```

Finding the 'best' Layout

...the operation best of Chapter 15.3.4 to find the 'best' layout of a document is generalized to act on a list of indentation-document pairs by combining it with the new representation function rep:

```
be w k z = best w k (rep z)      (hypothesis)
```

The new definition is directly derived from the old one:

```
best w k x                = be w k [(0,x)]

be w k []                 = Nil
be w k ((i,NIL):z)        = be w k z
be w k ((i,x :<> y) : z)  = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)   = s `Text` be w (k,+length s) z
be w k ((i,LINE) : z)     = i `Line` be w i z
be w k ((i.x :<|> y) : z) =
  better w k (be w k ((i.x) : z)) (be w k (i,y) : z))
```

Correctness

...of the equations of the previous slide can be shown by equational reasoning, e.g.:

Proposition 15.3.5.1

```
best w k x = be w k [(0,x)]
```

Proof by equational reasoning.

```
   best w k x
=  {0 is unit for nest}
   best w k (nest 0 x)
=  {nil is unit for <>}
   best w k (nest 0 x <> nil)
=  {Definition of rep, hypothesis}
   be w k [(0,x)]
```

□

Last but not least

...while the argument to best is represented using

- DOC

its result is represented using the formerly introduced type

- Doc

Hence, pretty can be defined as in Chapter 15.3.4:

```
pretty w x = layout (best w 0 x)
```

The functions layout, better, and fits, finally, remain unchanged.

Chapter 15.3.6

Utility Functions

Utility Functions (1)

...for recurringly occurring tasks, e.g.:

- Separating two documents by inserting a space:

```
x <+> y            = x <> text " " <> y
```

- Separating two documents by inserting a line break:

```
x </> y            = x <> line <> y
```

- Folding a document:

```
folddoc f []       = nil
folddoc f [x]      = x
folddoc f (x:xs) = f x (folddoc f xs)
```

- Advanced document folding:

```
spread             = folddoc (<+>)
stack              = folddoc (</>)
```

Utility Functions (2)

...as abbreviations of frequently occurring tasks, e.g.:

- An opening bracket, followed by an indented portion, followed by a closing bracket, abbreviated by `bracket`:

```
bracket l x r = group (text l <>
                       nest 2 (line <> x) <>
                       line <> text r)
```

- The 'right' layout strategy for trees of Chapter 15.2.3, abbreviated by `showBracket′`:

```
showBracket′ ts = bracket "[" (showTrees′ ts) "]"
```

- Taking a string, returning a document, where every line is filled with as many words as will fit (note: `words` is from the Haskell Standard Library), abbreviated by `fillwords`:

```
x <+/> y  = x <> (text " " :<|> line) <> y
fillwords = folddoc (<+/>) . map text . words
```

Utility Functions (3)

...abbreviations (cont'd):

- A variant of `fillwords` collapsing a list of documents to a single document by putting a space between two documents when this leads to a reaonsable layout, and a newline otherwise, abbreviated by `fill`:

```
fill []       = nil
fill [x]      = x
fill (x:y:zs) =
 (flatten x <+> fill (flatten y : zs)) :<|>
                            (x </> fill (y : zs)
```

Note: `fill` is copied from pretty printer library of Simon Peyton Jones, which extends the one of John Hughes.

Chapter 15.3.7

Printing XML-like Documents

Printing XML Documents

...enjoying a simplified XML syntax with elements, attributes, and text defined by:

```
data XML = Elt String [Att] [XML]
         | Txt String

data Att = Att String String
```

Utility Functions (1)

...for printing XML documents:

- Showing documents:

```
showXML x = folddoc (<>) (showXMLs x)
```

- Showing elements:

```
showXMLs (Elt n a []) =
  [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c) =
  [text "<" <> showTag n a <> text ">" <>
   showFill showXMLs c <>
   text "</" <> text n <> text ">"]
```

- Showing text:

```
showXMLs (Txt s) = map text (words s)
```

- Showing attributes:

```
showAtts (Att n v) =
  [text n <> text "=" <> text (quoted v)]
```

Utility Functions (2)

...for printing XML documents (cont'd):

- Adding quotes:

```
quoted s = "\"" ++ s ++ "\""
```

- Showing tags:

```
showTag n a   = text n <> showFill showAtts a
```

- Filling lines:

```
showFill f [] = nil
showFill f xs =
  bracket "" (fill (concat (map f xs))) ""
```

Example: 1st Layout of an XML Document

...for a maximum line width of 30 characters:

```
<p
  color="red" font="Times"
  size="10"
>
  Here is some
  <em> emphasized </em> text.
  Here is a
  <a
    href="http://www.eg.com/"
  > link </a>
  elsewhere.
</p>
```

Example: 2nd Layout of an XML Document

...for a maximum line width of 60 characters:

```
<p color="red" font="Times" size="10" >
  Here is some <em> emphasized </em> text. Here is a
  <a href="http://www.eg.com/" > link </a> elsewhere.
</p>
```

Example: 3rd Layout of an XML Document

...dropping the two occurrences of `flatten` in `fill` (cf. Chapter 15.3.6) leads to the following output:

```
<p color="red" font="Times" size="10" >
  Here is some <em>
    emphasized
  </em> text. Here is a <a
    href="http://www.eg.com/"
  > link </a> elsewhere.
</p>
```

...in the above layout start and close tags of the emphasis and anchor elements are crammed together with other text, rather than getting lines to themselves; it thus looks less 'beautiful.'

Chapter 15.4

The Prettier Printer Code Library

A Summary

...of the code of the

- performance-improved fully-fledged prettier printer.
- tree example.
- XML-documents example.

according to:

- Philip Wadler. A Prettier Printer. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

Chapter 15.4.1

The Prettier Printer

The Prettier Printer (1)

Defining operator priorities

```
infixr 5:<|>
infixr 6:<>
infixr 6 <>
```

Defining algebraic document types

```
data DOC = NIL
         | DOC :<> DOC
         | NEST Int DOC
         | TEXT String
         | LINE
         | DOC :<|> DOC

data Doc = Nil
         | String `Text` Doc
         | Int `Line` Doc
```

The Prettier Printer (2)

Defining basic operators algebraically

```
nil       = NIL
x <> y    = x :<> y
nest i x = NEST i x
text s    = TEXT s
line      = LINE
```

Layouting normal form documents

```
layout Nil           = ""
layout (s `Text` x) = s ++ layout x
layout (i `Line` x) = `\n`: copy i ` ` ++ layout x
copy i x             = [x | _ <- [1..i]]
```

The Prettier Printer (3)

Generating multiple layouts

```
group x = flatten x :<|> x
```

Flattening layouts

```
flatten NIL          = NIL
flatten (x :<> y)    = flatten x:<> flatten y
flatten (NEST i x)   = NEST i (flatten x)
flatten (TEXT s)     = TEXT s
flatten LINE         = TEXT " "
flatten (x :<|> y)   = flatten x
```

The Prettier Printer (4)

Ordering and comparing layouts

```
best w k x = be w k [(0,x)]

be w k [] = Nil
be w k ((i,NIL):z) = be w k z
be w k ((i,x :<> y) : z) = be w k ((i,x) : (i,y): z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z) = s `Text` be w (k+length s) z
be w k ((i,LINE) : z) = i `Line` be w i z
be w k ((i.x :<|> y) : z) =
 better w k (be w k ((i.x) : z)) (be w k (i,y) : z))

better w k x y = if fits (w-k) x then x else y

fits w x | w<0      = False
fits w Nil          = True
fits w (s `Text` x) = fits (w - length s) x
fits w (i `Line` x) = True
```

The Prettier Printer (5)

Printing documents prettily

```
pretty w x = layout (best w 0 x)
```

Defining utility functions

```
x <+> y          = x <> text " " <> y
x </> y          = x <> line <> y
x <+/> y         = x <> (text " " :<|> line) <> y

folddoc f []     = nil
folddoc f [x]    = x
folddoc f (x:xs) = f x (folddoc f xs)

spread           = folddoc (<+>)
stack            = folddoc (</>)

bracket l x r    =
 group (text l <> nest 2 (line <> x) <>
                                    line <> text r)
```

The Prettier Printer (6)

Defining utility functions (cont'd)

```
fillwords      = folddoc (<+/>) . map text . words

fill []        = nil
fill [x]       = x
fill (x:y:zs) =
 (flatten x <+> fill (flatten y : zs))
                            :<|> (x </> fill (y : zs)
```

Chapter 15.4.2

The Tree Example

The Tree Example (1)

Defining trees

```
data Tree = Node String [Tree]
```

Defining utility functions

```
showTree (Node s ts) =
 group (text s <> nest (length s) (showBracket ts))

showBracket [] = nil
showBracket ts =
 text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees [t]    = showTree t
showTrees (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

The Tree Example (2)

Defining utility functions (cont'd)

```
showTree′ (Node s ts) = text s <> showBracket′ ts

showBracket′ [] = nil
showBracket′ ts = bracket "[" (showTrees′ ts) "]"

showTrees′ [t]    = showTree t
showTrees′ (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

The Tree Example (3)

Defining a tree value for illustration

```
tree = Node "aaa"[ Node "bbbb"[ Node "ccc"[],
                                Node "dd"[]
                              ],
                   Node "eee"[],
                   Node "ffff"[ Node "gg"[],
                                Node "hhh"[],
                                Node "ii"[]
                              ]
                 ]
```

Defining two testing environments

```
testtree w  = putStr(pretty w (showTree tree))
testtree′ w = putStr(pretty w (showTree′ tree))
```

Chapter 15.4.3

The XML Example

The XML Example (1)

Defining the XML-like document format

```
data XML = Elt String [Att] [XML]
         | Txt String

data Att = Att String String
```

Defining utility functions

```
showXML x = folddoc (<>) (showXMLs x)

showXMLs (Elt n a []) =
 [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c) =
 [text "<" <> showTag n a <> text ">" <>
   showFill showXMLs c <>
    text "</" <> text n <> text ">"]
showXMLs (Txt s) = map text (words s)
```

The XML Example (2)

Defining utility functions (cont'd)

```
showAtts (Att n v) =
 [text n <> text "=" <> text (quoted v)]

quoted s = "\"" ++ s ++ "\""

showTag n a = text n <> showFill showAtts a

showFill f [] = nil
showFill f xs =
 bracket "" (fill (concat (map f xs))) ""
```

The XML Example (3)

Defining an XML-document value for illustration

```
xml =
 Elt "p"[Att "color" "red",
         Att "font" "Times",
         Att "size" "10"
        ] [Txt "Here is some",
           Elt "em" [] [Txt "emphasized"],
           Txt "text.",
           Txt "Here is a",
           Elt "a" [Att "href" "http://www.eg.com/"]
                   [Txt "link" ],
           Txt "elsewhere."
          ]
```

Defining a testing environment

```
testXML w = putStr (pretty w (showXML xml))
```

Chapter 15.5

Summary

Summary

...the pretty printer library proposed by John Hughes is widely recognized as a standard:

- John Hughes. The Design of a Pretty-Printer Library. In Johan Jeuring, Erik Meijer (Eds.), Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques. Springer-V., LNCS 925, 53-96, 1995.

...a variant of it is implemented in the Glasgow Haskell Compiler:

- Simon Peyton Jones. Haskell pretty-printer library. 1997. `www.haskell.org/libraries/#prettyprinting`

Why 'prettier' than 'pretty'?

...the pretty printer of John Hughes

- uses two operators for the horizontal and vertical concatenation of documents
  - one without a unit (vertical)
  - one with a right-unit but no left-unit (horizontal).

...the prettier printer of Philip Wadler can be considered an improvement of the pretty printer of John Hughes because it

- uses only one operator for document concatenation which
  - is associative.
  - has a left-unit and a right-unit.
- consists of about 30% less code.
- is about 30% faster.

In closing

...two notes on an early work on an imperative pretty printer by:

- Derek Oppen. Pretty-printing. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.

and a functional realization of it by:

- Olaf Chitil. Pretty Printing with Lazy Dequeues. In Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.

Chapter 15.6

References, Further Reading

Chapter 15: Further Reading (1)

- Manuel M.T. Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 13.1.2, Ausdrücke formatieren; Kapitel 13.2.1, Formatieren und Auswerten in erweiterter Version)

- Olaf Chitil. *Pretty Printing with Lazy Dequeues*. In Proceedings of the ACM SIGPLAN 2001 Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.

- John Hughes. *The Design of a Pretty-Printer Library*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 53-96, 1995.

Chapter 15: Further Reading (2)

- Derek Oppen. *Pretty-printing*. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.
- Tillmann Rendel, Klaus Ostermann. *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing*. In Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell 2010), 1-12, 2010.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 5, Writing a Library: Working with JSON Data – Pretty Printing a String, Fleshing Out the Pretty-Printing Library)

Chapter 15: Further Reading (3)

- Simon Peyton Jones. *Haskell pretty-printer library*. 1997. `www.haskell.org/libraries/#prettyprinting`
- Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 223-243, 2003.

Chapter 16

Functional Reactive Programming

Chapter 16.1

Motivation

Hybrid Systems

...are systems composed of

- continuous
- discrete

components.

Mobile Robots

...are special hybrid systems (or cyber-physical systems) from both a physical and logical perspective:

- Physically
  - Continuous components: Voltage-controlled motors, batteries, range finders,...
  - Discrete components: Microprocessors, bumper switches, digital communication,...
- Logically
  - Continuous notions: Wheel speed, orientation, distance from a wall,...
  - Discrete notions: Running into another object, receiving a message, achieving a goal,...

In this chapter

...designing and implementing two

- imperative-style languages for controlling robots

Beyond the concrete application, this provides two examples of

- domain specific language (DSL)

and an application of the type constructor classes

- Monad
- Arrow
- Functor

Note, the languages aim at simulating robots in order to allow running simulations at home without having to buy (possibly expensive) robots first.

Reading

...for Chapter 16.2 (using monads):

- Paul Hudak. The Haskell School of Expression – Learning Functional Programming through Multimedia. Cambridge University Press, 2000. (Chapter 19, An Imperative Robot Language)

...for Chapter 16.3 (using arrows):

- Paul Hudak, Antony Courtney, Herik Nilsson, John Peterson. Arrows, Robots, and Functional Reactive Programming. Summer School on Advanced Functional Programming 2002, Springer-V., LNCS 2638, 159-187, 2003.

Note: Chapter 16.2 and 16.3 are independent and do not build upon each other.

Chapter 16.2

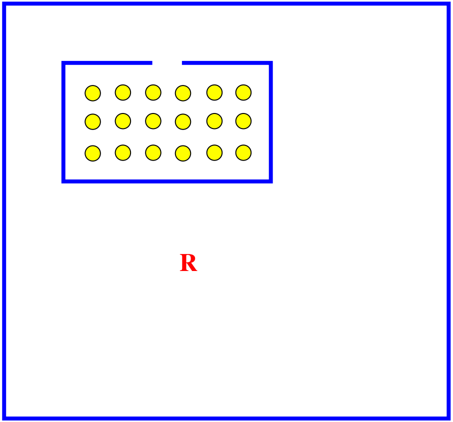
An Imperative Robot Language

Chapter 16.2.1

The Robot's World

The Robot's World

...a two-dimensional grid surrounded by walls, with rooms having doors, and gold coins as treasures!



In more detail

...the robot's world is

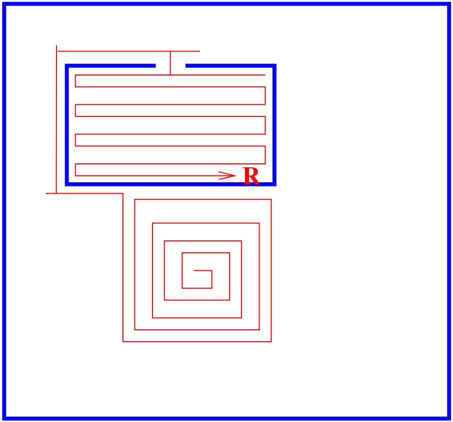
- a finite two-dimensional grid of square form
  - equipped with walls
  - possibly forming rooms, possibly having doors
  - with gold coins placed on some grid points

The preceding example shows

- a robot's world with one room, an open door, full of gold: Eldorado!
- a robot sitting in the centre of the world ready for exploring it!

The Robot's Mission

...exploring the world, collecting treasures, leaving footprints!



In more detail

...the robot's mission is

- to explore its world, to collect the treasures in it, and to leave footprints of its exploration, i.e., to
  - strolling and searching through its world, e.g., following the path way of an outward-oriented spiral.
  - picking up the gold coins it finds on its way and saving them in its pocket.
  - dropping gold coins at some (other) grid points.
  - marking its way with differently colored pens.

Objective

...developing an imperative-like robot language allowing to write programs, which advise a robot how to explore and shape its world!

E.g., programs such as:

```
(1) drawSquare =
     do penDown
        move
        turnRight
        move
        turnRight
        move
        turnRight
        move
```

```
(2) moveToWall =
     while (isnt blocked)
      do move
```

```
(3) getCoinsToWall =
     while (isnt blocked) $
      do move
         checkAndPickCoin
```

In more detail

...assuming that Robot is a monad:

```
newtype Robot a = Rob...
instance Monad Robot where...

drawSquare =
 do penDown      (penDown :: Robot () / pen ready to write)
    move         (move :: Robot () / moving one space for-
    turnRight                        ward)
    move
    turnRight    (turnRight: Robot () / turn 90 degrees
    move                                    clock-wise)
    turnRight
    move
```

Note, for the robot monad, operation (>>) is relevant!

The Implementation Environment

...required modules:

```
module Robot where
  import Array
  import List
  import Monad
  import SOEGraphics
  import Win32Misc (timeGetTime)
  import qualified GraphicsWindows as GW (getEvent)
```

Note:

- `Graphics`, `SOEGraphics` are two commonly used graphics libraries being Windows compatible.
- Double-check the SOE homepage at `haskell.org/soe` regarding the availability of the modules `SOEGraphics` and `GraphicsWindows`.

Chapter 16.2.2

Modelling the Robot's World

Modelling the World

...the robots live and act in a 2-dimensional grid.

Positions are given by their x and y coordinates:

```
type Position = (Int,Int)
```

Directions a robot can face or head to:

```
data Direction = North | East | South | West
                      deriving (Eq, Show, Enum)
```

World, a two-dimensional grid as `Array`-type:

```
type Grid = Array Position [Direction]
```

Chapter 16.2.3

Modelling Robots

Modelling Robots

...by their internal states, which are characterized by 6 values:

1. Robot position
2. Robot orientation
3. Pen status (up or down)
4. Pen color
5. Treasure map
6. Number of coins in the robot's pocket

Note, the grid does not change and is thus not part of a robot (state).

Modelling Internal Robot States

...as an algebraic product type:

```
data RobotState = RState { position :: Position
                         , facing :: Direction
                         , pen :: Bool
                         , color :: Color
                         , treasure :: [Position]
                         , pocket :: Int
                         } deriving Show
```

where the number of coins at a position is given by the number of its occurrences in `treasure`, and `Color` defines the set of possible pen colors:

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
            deriving (Eq, Ord, Bounded, Enum,
                      Ix, Show, Read)
```

Note

...the definition of RobotState takes advantage of Haskell's field-label (or record) syntax: The field labels (position, facing, pen, color, treasure, pocket) offer

- access to state components by names instead of position without requiring specific selector functions.

This advantage would have been lost defining robot states equivalently but without field-label syntax as in:

```
data RobotState = RState
                    Position
                    Direction
                    Bool
                    Color
                    [Position]
                    Int deriving Show
```

Illustrating Field-label Syntax Usage (1)

...generating, modifying, and accessing values of robot-state components.

Example 1: Generating field values

The definition

```
s1 = RState { position = (0,0)
            , facing   = East
            , pen      = True
            , color    = Green
            , treasure = [(2,3),(7,9),(12,42)]
            , pocket   = 2
            } :: RobotState
```

is equivalent to:

```
s2 = RState (0,0) East True Green
            [(2,3),(7,9),(12,42)] 2 :: RobotState
```

Illustrating Field-label Syntax Usage (2)

Example 2: Modifying field values

```
s3 = s2 { position = (22,43), pen = False }
 ->> RState { position = (22,43)
            , facing = East
            , pen = False
            , color = Green
            , treasure = [(2,3),(7,9),(12,42)]
            , pocket = 2
            } :: RobotState
```

Example 3: Accessing field values

```
position s1   ->>  (0,0)
treasure s3   ->>  [(2,3),(7,9),(12,42)]
color    s3   ->>  Green
```

Example 4: Using field names in patterns

```
jump (RState { position = (x,y) }) = (x+2,y+1)
```

Benefits and Advantages

...of using field-label syntax:

- It is more 'informative' (due to field names).
- The order of fields gets irrelevant, e.g., the definition of:

```
s4 = RState { position = (0,0)
            , pocket   = 2
            , pen      = True
            , color    = Green
            , treasure = [(2,3),(7,9),(12,42)]
            , facing   = East
            } :: RobotState
```

  is equivalent to the robot state defined by `s1`.

Chapter 16.2.4

Modelling Robot Commands as State Monad

Modelling Robot Commands

...by Robot, a 1-ary type constructor, defined by:

```
newtype Robot a =
 Rob (RobotState -> Grid -> Window
                              -> IO (RobotState,a))
```

allows making Robot an instance of type class Monad (matching the pattern of a state monad by concepually considering the Grid argument part of the state):

```
instance Monad Robot where
 Rob sf0 >>= f = Rob $ \s0 g w ->
                       do (s1,a1) <- sf0 s0 g w
                          let Rob sf1 = f a1
                          (s2,a2) <- sf1 s1 g w
                          return (s2,a2)
 return a      = Rob (\s _ _ -> return (s,a))
```

Note

- $ can be replaced by parentheses:

```
instance Monad Robot where
 Rob sf0 >>= f = Rob (\s0 g w ->
                       do (s1,a1) <- sf0 s0 g w
                          let Rob sf1 = f a1
                          (s2,a2) <- sf1 s1 g w
                          return (s2,a2))
 return a      = Rob (\s _ _ -> return (s,a))
```

- the Grid argument in

```
newtype Robot a =
 Rob (RobotState -> Grid -> Window
                                  -> IO (RobotState,a))
```

  can conceptually be considered a 'read- only' part of a robot state; the Window argument allows specifying the window, in which the graphics is displayed.

Chapter 16.2.5

The Imperative Robot Language

IRL: The Imperative Robot Language

Key insight:

- Taking state as input
- Possibly querying the state in some way
- Returning a possibly modified state

...reveals the imperative nature of IRL commands.

Utility Functions

...not intended (except of at) for direct usage by an IRL programmer.

- Direction commands:

```
right, left :: Direction -> Direction
right d = toEnum (succ (mod (fromEnum d) 4))
left d  = toEnum (pred (mod (fromEnum d) 4))

at :: Grid -> Position -> [Direction]
at = (!)
```

- Supporting functions for updating and querying states:

```
updateState :: (RobotState -> RobotState)
                                      -> Robot ()
updateState u = Rob (\s _ _ -> return (u s, ()))

queryState :: (RobotState -> a) -> Robot a
queryState q = Rob (\s _ _ -> return (s, q s))
```

Recalling the Definition of Type Class Enum

...of the Standard Prelude:

```
class Enum a where
 succ, pred     :: a -> a
 toEnum         :: Int -> a
 fromEnum       :: a -> Int
 enumFrom       :: a -> [a]              -- [n..]
 enumFromThen   :: a -> a -> [a]         -- [n,n′..]
 enumFromTo     :: a -> a -> [a]         -- [n..m]
 enumFromThenTo :: a -> a -> a -> [a]    -- [n,n′..m]

 succ                   = toEnum . (+1) . fromEnum
 pred                   = toEnum . (subtract 1) . fromEnum
 enumFrom x             = map toEnum [fromEnum x..]
 enumFromThen x y       = map toEnum [fromEnum x, fromEnum y..]
 enumFromTo x y         = map toEnum [fromEnum x..fromEnum y]
 enumFromThenTo x y z = map toEnum [fromEnum x,
                                    fromEnum y..fromEnum z]

 toEnum, fromEnum =    ...implementation is type-dependent
```

Recalling the Usage of Type Class Enum

The following 'equalities' hold:

$$
\begin{array}{lcl}
\texttt{enumFrom n} & \widehat{=} & \texttt{[n..]} \\
\texttt{enumFromThen n n}' & \widehat{=} & \texttt{[n,n}'\texttt{..]} \\
\texttt{enumFromTo n m} & \widehat{=} & \texttt{[n..m]} \\
\texttt{enumFromThenTo n n}'\texttt{ m} & \widehat{=} & \texttt{[n,n}'\texttt{..m]}
\end{array}
$$

Example:

```
data Color = Red | Orange | Yellow | Green
             | Blue | Indigo | Violet deriving Enum

[Red..Green]      ->>  [Red, Orange, Yellow, Green]
[Red, Yellow..]   ->>  [Red, Yellow, Blue, Violet]
fromEnum Blue     ->>  4
toEnum 3          ->>  Green
```

IRL Commands for Robot Orientation

...by updating the internal robot state.

- Turn right:

```
turnLeft :: Robot ()
turnLeft =
 updateState (\s -> s {facing = left (facing s)})
```

- Turn left:

```
turnRight :: Robot ()
turnRight =
 updateState (\s -> s {facing = right (facing s)})
```

- Turn to:

```
turnTo :: Direction -> Robot ()
turnTo d = updateState (\s -> s {facing = d})
```

- Facing what direction?

```
direction :: Robot Direction
direction =  queryState facing
```

IRL Command for Blockade Checking

- Motion blocked in direction currently facing?

```
blocked :: Robot Bool
blocked =
 Rob $ \s g _ ->
  return (s, facing s `notElem` (g `at` position s))
```

with `notElem` from the Standard Prelude.

IRL Commands for Motion

- Moving forward one space if not blocked:

```
move :: Robot ()
move =
 cond1 (isnt blocked)
  (Rob $ \s _ w -> do
    let newPos = movePos (position s) (facing s)
    graphicsMove w s newPos
    return (s {position = newPos}, ())
   )
```

- Moving forward one space in direction of:

```
movePos :: Position -> Direction -> Position
movePos (x,y) d = case d of North -> (x,y+1)
                            South -> (x,y-1)
                            East  -> (x+1,y)
                            West  -> (x-1,y)
```

IRL Commands for Pen Usage

- Choose pen color for writing:

```
setPenColor :: Color -> Robot ()
setPenColor c = updateState (\s -> s {color = c})
```

- Pen down to start writing:

```
penDown :: Robot ()
penDown = updateState (\s -> s {pen = True})
```

- Pen up to stop writing:

```
penUp :: Robot ()
penUp = updateState (\s -> s {pen = False})
```

IRL Commands for Coin Handling (1)

- At position with coin according to treasure map?

```
onCoin :: Robot Bool
onCoin = queryState (\s ->
               position s `elem` treasure s)
```

- Pick coin:

```
pickCoin :: Robot ()
pickCoin =
 cond1 onCoin
  (Robot $ \s _ w ->
     do eraseCoin w (position s)
        return (s {treasure =
                   position s `delete` treasure s,
                  pocket = pocket s+1}, ())
  )
```

IRL Commands for Coin Handling (2)

- How many coins currently in pocket?

```
coins :: Robot Int
coins = queryState pocket
```

- Drop coin, if there is at least one in the pocket:

```
dropCoin :: Robot ()
dropCoin =
 cond1 (coins >* return 0)
  (Robot $ \s _ w ->
     do drawCoin w (position s)
        return (s {treasure =
                     position s : treasure s,
                  pocket = pocket s-1}, ())
  )
```

Utility Functions for Logic and Control (1)

- Conditionally performing commands:

```
cond :: Robot Bool -> Robot a
                   -> Robot a -> Robot a
cond p c a = do pred <- p
                if pred then c else a

cond1 p c = cond p c (return ())
```

- Performing commands while some condition is met:

```
while :: Robot Bool -> Robot () -> Robot ()
while p b = cond1 p (b >> while p b)
```

- Connecting commands 'disjunctively:'

```
(||*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 ||* b2 = do p <- b1
               if p then return True
                    else b2
```

Utility Functions for Logic and Control (2)

- Connecting commands 'conjunctively:'

```
(&&*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 &&* b2 = do p <- b1
               if p then b2
                    else return False
```

- Lifting negation to commands:

```
isnt :: Robot Bool -> Robot Bool
isnt =  liftM not
```

- Lifting comparisons to commands:

```
(>*) :: Robot Int -> Robot Int -> Robot Bool
(>*) =  liftM2 (>)

(<*) :: Robot Int -> Robot Int -> Robot Bool
(<*) =  liftM2 (<)
```

Recalling the Definitions of the Lift Operators

...the higher-order lift operations `liftM` and `liftM2` are defined in the library `Monad` (as well as `liftM3`, `liftM4`, and `liftM5`):

```
liftM :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f =  \a -> do a' <- a
                    return (f a')

liftM2 :: (Monad m) => (a -> b -> c)
                             -> (m a -> m b -> m c)
liftM2 f =  \a b -> do a' <- a
                       b' <- b
                       return (f a' b')
```

Note

The implementations of

- `isnt`, `(>*)`, and `(<*)` are based on `liftM` and `liftM2`, thereby avoiding the usage of special lift functions.
- `(||*)` and `(&&*)` are not based on `liftM2`, thereby avoiding (unnecessary) strictness in their second arguments.

Illustrating the Usage of cond and cond1

...moving the robot one space forward if it is not blocked; moving it one space to the right if it is.

An implementation using

- cond:

```
evade :: Robot ()
evade = cond blocked
             (do turnRight
                 move)
             move
```

- cond1:

```
evade′ :: Robot ()
evade′ = do cond1 blocked turnRight
            move
```

Moving in a Spiral

...an example of an advanced IRL program:

```
spiral :: Robot ()
spiral = penDonw >> loop 1
 where loop n =
        let twice = do turnRight
                       moven n
                       turnRight
                       moven n
        in con blocked
            (twice >> turnRight >> moven n)
            (twice >> loop (n+1))

moven :: Int -> Robot ()
moven n = mapM . (const move) [1..]
```

Chapter 16.2.6

Defining a Robot's World

The Robot's World: Preliminary Definitions

The robots' world is a grid of type `Array`:

```
type Grid = Array Position [Direction]
```

Grid points can be accesssed using:

```
at :: Grid -> Position -> [Direction]
at =  (!)
```

Defining the Initial World g0 (1)

The size of the initial grid world g0 is given by:

```
size :: Int
size =  20
```

with the grid world's

- centre at: `(0,0)`
- corners at:

```
   (-size,size)        (size,size)
((-size),(-size))   (size,(-size))
```

Defining the Initial World g0 (2)

..inner, border, and corner points of world g0 are characterized by the directions of motion they allow:

- Inner points of g0 allow moving toward:
  ```
  interior = [North, South, East, West]
  ```
- Border points at the north, east, south, and west border allow moving toward:
  ```
  nb  = [South, East, West]        (nb: north border)
  eb  = [North, South, West]
  sb  = [North, East, West]
  wb  = [North, South, East]       (wb: west border)
  ```
- Corner points at the northwest, northeast, southeast, and southwest corner allow moving toward:
  ```
  nwc = [South, East]       (nwc: northwest corner)
  nec = [South, West]
  sec = [North, West]
  swc = [North, East]       (swc: southwest corner)
  ```

Defining the Initial World g0 (3)

...all grid points, i.e., inner and border grid points can thus be enumerated using list comprehension, which allows to define the initial world grid g0 as follows:

```
g0 :: Grid
g0 = array ((-size, -size), (size, size))
      ([((i, size), nb)   | i <- r ] ++
      ([((i, -size), sb)  | i <- r ] ++
      ([((size, i), eb)   | i <- r ] ++
      ([((-size, i), wb)  | i <- r ] ++
      ([((size, i), eb)   | i <- r ] ++
      ([((i,j), interior) | i <- r, j <- r ] ++
      ([((size, size), nec), ((size, -size), sec),
        ((-size, size), nwc),
        ((-size, -size), swc)])
     where r = [1-size..size-1]
```

Building World g1 from World g0

...by erecting a west/east-oriented wall leading from (-5,10) to (5,10):

```
g1 :: Grid
g1 =  g0 // mkHorWall (-5) 5 10
```

where (//) is the Array library function (cf. Chapter 7.2):

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

Recalling the (//) Function

...of the Array library:

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

and illustrating its usage: To this end, let:

```
colors :: Array Int Color
colors = array (0,7)
          [(0,Black),(1,Blue),(2,Green),(3,Cyan),
           (4,Red),(5,Magenta),(6,Yellow),
           (7,White)]
```

then:

```
colors // [(0,White),(7,Black)]
 ->> array (0,7) [(0,White),(1,Blue),(2,Green),(3,Cyan),
                  (4,Red),(5,Magenta),(6,Yellow),
                  (7,Black)] :: Array Int Color
```

swaps the 'black' und 'white' entries in colors.

Note

Type Color is defined as in the

- Graphics library:

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
            deriving (Eq, Ord, Bounded, Enum,
                      Ix, Show, Read)
```

Equivalently but more concisely we could have defined

- colors by:

```
colors :: Array Int Color
colors = array (0,7) (zip [0..7] [Black..White])
```

Utility Functions for Building Walls

Building walls horizontally (west/east-oriented, leading from `(x1,y)` to `(x2,y)`):

```
mkHorWall :: Int -> Int -> Int -> [(Position,[Direction])]
mkHorWall x1 x2 y =
 [((x,y), nb) | x <- [x1..x2]] ++
    [((x,y+1), sb) | x <- [x1..x2]]
```

Building walls vertically (north/south-oriented, leading from `(x,y1)` to `(x,y2)`):

```
mkVerWall :: Int -> Int -> Int -> [(Position,[Direction])]
mkVerWall y1 y2 x =
 [((x,y), eb) | y <- [y1..y2]] ++
    [((x+1,y), wb) | y <- [y1..y2]]
```

Utility Functions for Building Rooms

...naively, rooms could be built using mkHorWall and mkVerWall straightforwardly:

```
mkBox :: Position -> Position
                    -> [(Position, [Direction])]
mkBox (x1, y1) (x2, y2) =
 mkHorWall (x1+1) x2 y1 ++ mkHorWall (x1+1) x2 y2 ++
 mkVerWall (y1+1) y2 x1 ++ mkVerWall (y1+1) y2 x2
```

This, however, creates two field entries for each of the four inner corners causing their values undefined after the call is finished (cf. Chapter 7.2).

This problem can elegantly be overcome by using the Array library operation accum (cf. Chapter 7.2) in combination with mkBox.

Recalling the accum Function

...of the `Array` library:

```
accum :: (Ix a) => (b -> c -> b)
                 -> Array a b -> [(a,c)] -> Array a b
```

As discussed in Chapter 7.2, `accum`

- is quite similar to `(//)`.
- in case of replicated entries the function of the first argument is applied for resolving conflicts.
- the `intersect` function of the `List` library is appropriate for this in the case of our example, e.g.:

  ```
  [South, East, West] `intersect`
      [North, South, West] ->> [South, West]
  ```

  represents the northeast corner.

Building World g2 from World g0

...by building a room with its lower left and upper right corner at positions (-10,5) and (-5,10), respectively:

```
g2 :: Grid
g2 = accum intersect g0 (mkBox (-15,8) (2,17))
```

using accum, intersect, and mkBox.

Building World g3 from World g2

...by adding a door (to the middle of the top wall of the room)

```
g3 :: Grid
g3 = accum union g2 [((-7,17), interior),
                     ((-7,18), interior)]
```

using accum, union, and interior.

Chapter 16.2.7

Robot Graphics: Animation in Action

Objective of Animation

...drawing the world the robot lives in and then showing the robot running around (at some predetermined rate) accomplishing its mission:

- Drawing lines if the pen is down.
- Picking up coins.
- Dropping coins, letting them thereby appear in possibly other locations.

This requires to incrementally update the drawn and displayed graphics, which will be achieved by means of the operations of the `Graphics` library.

Updating the Graphics Incrementally

...key for incrementally updating the displayed world the Graphics library operation drawInWindowNow:

```
drawInWindowNow :: Window -> Color
                         -> Point -> Point -> IO ()
```

which draws the updated graphics immediately after any changes, and can be used, e.g., for drawing lines:

```
drawLine :: Window -> Color
                         -> Point -> Point -> IO ()
drawLine w c p1 p2 =
  drawInWindowNow w (withColor c (line p1 p2))
```

Note

...in order to work properly, the incremental update of the world must be organized such that the

- absence of interferences of graphics actions

is ensured.

This is achieved by assuming:

1. Grid points are 10 pixels apart.
2. Walls are drawn halfway between grid points.
3. The robot pen draws lines directly from one grid point to the next.
4. Coins are drawn as yellow circles just above and to to the left of each grid point.
5. Coins are erased by drawing black circles over the yellow ones which are already there.

Defining Top-level Constants

...for dealing with the preceding assumptions.

Half the distance between grid points:

```
d :: Int
d = 5
```

Color of walls and coins:

```
wc, cc :: Color
wc = Blue
cc = Yellow
```

Window size:

```
xWin, yWin :: Int
xWin = 600
yWin = 500
```

Defining Utility Functions (1)

Drawing grids:

```
drawGrid :: Window -> Grid -> IO ()
drawGrid w wld =
 let (low@(xMin,yMin),hi@(xMax,yMax)) = bounds wld
     (x1,y1)                          = trans low
     (x2,y2)                          = trans hi
 in
  do drawLine w wc (x1-d,y1+d) (x1-d,y2-d)
     drawLine w wc (x1-d,y1+d) (x1+d,y2+d)
     sequence_ [drawPos w (trans (x,y)) (wld `at` (x,y))
               | x <- [xMin..xMax], y <- [yMin..yMax]]
```

Defining Utility Functions (2)

Used by drawGrid:

```
drawPos :: Window -> Point -> [Direction] -> IO ()
drawPos x (x,y) ds =
 do if North `notElem` ds
       then drawLine w wc (x-d,y-d) (x+d,y-d)
       else return ()
    if East `notElem` ds
       then  drawLine w wc (x+d,y-d) (x+d,y+d)
       else return ()
```

Used by drawGrid, from the Array library:

```
bounds :: Ix a => Array a b -> (a,a)
 -- yields the bounds of its array argument
```

Defining Utility Functions (3)

Dropping and drawing coins:

```
drawCoins :: Window -> RobotState -> IO ()
drawCoins w s = mapM_ (drawCoin w) (treasure s)

drawCoin :: Window -> Position -> IO ()
drawCoin w p =
 let (x,y) = trans p
 in drawInWindowNow w
    (withColor cc (ellipse (x-5,y-1) (x-1,y-5)))
```

Erasing coins:

```
eraseCoin :: Window -> Position -> IO ()
eraseCoin w p =
 let (x,y) = trans p
 in drawInWindowNow w
     (withColor Black (ellipse (x-5,y-1) (x-1,y-5)))
```

Defining Utility Functions (4)

Drawing robot moves:

```
graphicsMove :: Window -> RobotState
                               -> Position -> IO ()
graphicsMove w s newPos =
 do if pen s
       then drawLine w (color s) (trans (position s))
                                  (trans newPos)
       else return ()
    getWindowTick w

trans        :: Position -> Point
trans (x,y) = (div xWin 2+2*d*x, div yWin 2-2*d*y)
```

Causing a short delay after each robot move

```
getWindowTick :: Window -> IO ()
```

Running IRL Programs: The Top-level Prg. (1)

...putting it all together.

Running an IRL program:

```
runRobot :: Robot () -> RobotState -> Grid -> IO ()
runRobot (Robot sf) s g =
 runGraphics $
 do w <- openWindowEx "Robot World" (Just (0,0))
          (Just (xWin, yWin)) drawGraphic (Just 10)
    drawGrid w g
    drawCoins w s
    spaceWait w
    sf s g w
    spaceClose w
```

Running IRL Programs: The Top-level Prg. (2)

Intuitively, runRobot

- opens a window
- draws a grid
- draws the coins
- waits for the user to hit the spacebar
- continues running the program with starting state s and grid g
- closes the window when execution is complete and the spacebar is pressed again.

where spaceWait provides the user with progress control by awaiting the user's pressing the spacebar:

```
spaceWait   :: Window -> IO ()
spaceWait w = do k <- getKey w
                 if k == ' ' then return ()
                             else spaceWait w
```

Animation in Action (1)

...the grids g0 through g3 can now be used to run IRL programs with.

1) Fixing s0 as a suitable starting state:

```
s0 :: RobotState
s0 = RobotState { position = (0,0)
                , pen = False
                , color = Red
                , facing = North
                , treasure = tr
                , pocket = 0
                }
```

2) Placing 'treasure' (all coins are placed inside the room in grid g3):

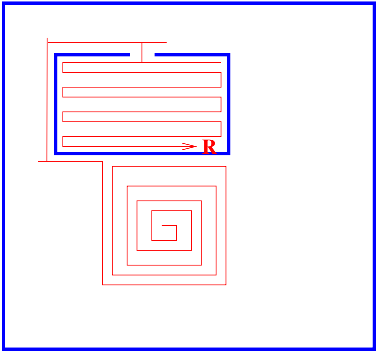
```
tr :: [Position]
tr = [(x,y) | x <- [-13,-11..1], y <- [9,11..15]]
```

Animation in Action (2)

3) Running the 'spiral' program with s0, g0:

```
main = runRobot spiral s0 g0
```

...leads to the 'spiral' example shown for illustration at the beginning of this chapter:



Chapter 16.3

Robots on Wheels

Outline

...we consider and define a simulation of

- mobile robots (called Simbots)

using functional reactive programming.

The implementation will make use of the type class

- `Arrow`

which is another example of a type constructor class generalizing the concept of a monad.

Chapter 16.3.1

The Setting

The Configuration of Mobile Robots (1)

...is assumed to be as follows:

"Robots are differential drive robots having two wheels that are each driven by an independent motor. The relative velocity of these two wheels governs the turning rate of the robot. If the velocities are identical, the robot will go straight.

A robot has several kinds of sensors. Among these, (1) a bumper switch to detect when the robot gets 'stuck' because of being blocked by something, (2) a range finder to determine the nearest object in any given direction (in the following it is assumed that there are four independent range finders that only look forward, backward, left and right; the range finder will thus only be queried at these four angles), (4) an animate object tracker that gives the current position of all other robots and possibly those of some free-moving balls that are within a certain distance from the robot.

The Configuration of Mobile Robots (2)

This object tracker can be thought of as *modelling either a visual subsystem that can 'see' these objects, or a communication subsystem through which the robots and balls share each other's coordinates. Some further capabilities will be introduced as need occurs.*

Last but not least, each robot has a unique ID."

The Application Scenario: Robot Soccer

...the overall task:

"Write a program to play 'robocup soccer' as follows:

Use wall segments to create two goals at either end of the field.

Decide on a number of players on each team and write generic controllers, such as one for a goalkeeper, one for attack, and one for defense.

Create an initial world where the ball is at the center mark, and each of the players is positioned strategically while being on-side (with the defensive players also outside of the center circle. Each team may use the same controller, or different ones."

Code for ‘Robots on Wheels’

...can be down-loaded at the Yampa homepage at

http://www.haskell.org/yampa

In the following we will consider essential code snippets.

Chapter 16.3.2

Modelling the Robots' World

Signal Functions, Signals, and Simbots

Signal functions are

- signal transformers, i.e., functions mapping signals to signals,
- of type `SF`, a 2-ary type constructor defined in Yampa, which is an instance of type constructor class `Arrow`.

Yampa provides

- a number of primitive signal functions and a set of special composition operators (or combinators) for constructing (more) complex signal functions from simpler ones.

Signals are no

- first-class values in Yampa but can only be manipulated by means of signal functions to avoid time- and space-leaks (abstract data type).

Simbot is a short hand for simulated robot.

Modelling Time, Signals, and Signal Functions

SF is an instance of class Arrow:

```
type Time      = Double
type Signal a~ = Time -> a
type SF a b    = Signal a -> Signal b
```

Intuitively: SF-values are signal transformers resp. signal functions (thus the type name SF).

Modelling Simbots

```
type RobotType = String
type RobotId   = Int

type SimbotController =
      SimbotProperties -> SF SimbotInput SimbotOutput

Class HasRobotProperties i where
 rpType      :: i -> RobotType    -- Type of robot
 rpId        :: i -> RobotId      -- Identity of robot
 rpDiameter :: i -> Length       -- Distance between wheels
 rpAccMax    :: i -> Acceleration -- Max translational acc
 rpWSMax     :: i -> Speed        -- Max wheel speed
```

Modelling the World

```
type WorldTemplate = [ObjectTemplate]

data ObjectTemplate =
    OTBlock    otPos  :: Position2  -- Square obstacle
  | OTVWall    otPos  :: Position2  -- Vertical wall
  | OTHWall    otPos  :: Position2  -- Horizontal wall
  | OTBall     otPos  :: Position2  -- Ball
  | OTSimbotA  otRId  :: RobotId,   -- Simbot A robot
               otPos  :: Position2,
               otHdng :: Heading
  | OTSimbotB  otRId  :: RobotId,   -- Simbot B robot
               otPos  :: Position2,
               otHdng :: Heading
```

Chapter 16.3.3

Classes of Robots

Types of Robots

...usually, there are different types of robots

- differing in their features (2 wheels, 3 wheels, camera, sonar, speaker, blinker, etc.)

The type of a robot is fixed by its

- input and output types

which are encoded in input and output classes together with the functions operating on the class elements.

Input Classes (1)

...and functions operating on their elements:

```
data BatteryStatus = BSHigh | BSLow | BSCritical
                     deriving (Eq, Show)

class HasRobotStatus i where
 -- Current battery status
 rsBattStat :: i -> BatteryStatus
 -- Currently stuck or not stuck
 rsIsStuck  :: i -> Bool

-- Derived event sources:
rsBattStatChanged  :: HasRobotStatus i =>
                           SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i => SF i (Event ())
rsBattStatCritical :: HasRobotStatus i => SF i (Event ())
rsStuck            :: HasRobotStatus i => SF i (Event ())
```

Input Classes (2)

```
class HasOdometry where
 -- Current position
 odometryPosition :: i -> Position2
 -- Current heading
 odometryHeading  :: i -> Heading

class HasRangeFinder i where
 rfRange    :: i -> Angle -> Distance
 rfMaxRange :: i -> Distance

-- Derived range finders:
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance
```

Input Classes (3)

```
class HasAnimateObjectTracker i where
 aotOtherRobots :: i -> [(RobotType, Angle, Distance)]
 aotBalls       :: i -> [(Angle, Distance)]

class HasTextualConsoleInput i where
 tciKey :: i -> Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i =>
                   Maybe Char -> SF i (Event Char)
tciKeyDown    :: HasTextualConsoleInput i =>
                   SF i (Event Char)
```

Output Classes

...and functions operating on their elements:

```
class MergeableRecord o => HasDiffDrive o where
 -- Brake both wheels
 ddBrake :: MR o
 -- Set wheel velocities
 ddVelDiff :: Velocity -> Velocity -> MR o
 -- Set velocities and rotation
 ddVelTR :: Velocity -> RotVel -> MR o

class MergeableRecord o =>
  HasTextConsoleOutput o where
    tcoPrintMessage :: Event String -> MR o
```

Chapter 16.3.4

Robot Simulation in Action

Typical Structure of a Robot Control Program

```
module MyRobotShow where

  import AFrob
  import AFrobRobotSim

  main :: IO ()
  main = runSim (Just world) rcA rcB

  world :: WorldTemplate
  world = ...

  -- controller for simbot A
  rcA :: SimbotController
  rcA = ...

  -- controller for simbot B
  rcB :: SimbotController
  rcB = ...
```

Robot Simulation in Action

Running a robot simulation:

```
runSim :: Maybe WorldTemplate
                -> SimbotController
                     -> SimbotController -> IO ()
```

Simbot controllers:

```
rcA :: SimbotController
rcA rProps =
  case rrpId rProps of
    1 -> rcA1 rProps
    2 -> rcA2 rProps
    3 -> rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
```

Chapter 16.3.5

Examples

Robot Actions: Control Programs (1)

A stationary robot:

```
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)
```

A blind robot moving at constant speed:

```
rcBlind1 _ =
 constant (mrFinalize $ ddVelDiff 10 10)
```

A blind robot moving at half the maximum speed:

```
rcBlind2 rps =
 let max = rpWSMax rps
 in constant (mrFinalize $
                  ddVelDiff (max/2) (max/2))
```

Robot Actions: Control Programs (2)

A robot rotating at a pre-given speed:

```
rcTurn :: Velocity -> SimbotController
rcTurn vel rps =
  let vMax = rpWSMax rps
      rMax = 2 * (vMax - vel) / rpDiameter rps
  in constant (mrFinalize $ ddVelTR vel rMax)
```

Chapter 16.4

Summary

The Origins

...of functional reactive programming (FRP) can be traced back to functional reactive animation (FRAn):

- Conal Elliot, Paul Hudak. Functional Reactive Animation. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.
- Conal Elliot. Functional Implementations of Continuous Modeled Animation. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.

Seminal Works

...on functional reactive programming (FRP):

- Zhanyong Wan, Paul Hudak. Functional Reactive Programming from First Principles. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Languages Design and Implementation (PLDI 2000), ACM Press, 2000.
- John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. Yale FRP User's Manual. Department of Computer Science, Yale University, January 2001. http://www.haskell.org/frp/manual.html
- Henrik Nilsson, Antony Courtney, John Peterson. Functional Reactive Programming, Continued. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.

Applications of FRP (1)

...on Functional Reactive Robotics (FRob):

- Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.
- John Peterson, Gregory Hager, Paul Hudak. A Language for Declarative Robotic Programming. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.

Applications of FRP (2)

...on Functional Animation Languages (FAL):

- Paul Hudak. The Haskell School of Expression – Learning Functional Programming through Multimedia. Cambridge University Press, 2000. (Chapter 15, A Module of Reactive Animations)

...on Functional Vision Systems (FVision):

- Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. Prototyping Real-Time Vision Systems: An Experiment in DSL Design. In Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), 484-493, 1999.

...on Functional Reactive User Interfaces (FRUIt):

- Antony Courtney, Conal Elliot. Genuinely Functional User Interfaces. In Proceedings of the 2001 Haskell Workshop, September 2001.

Applications of FRP (3)

...towards Real-Time FRP (RT-FRP):

- Zhanyong Wan, Walid Taha, Paul Hudak. Real-Time FRP. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
- Zhanyong Wan. Functional Reactive Programming for Real-Time Embedded Systems. PhD thesis. Department of Computer Science, Yale University, December 2002.

...towards Event-Driven FRP (ED-FRP):

- Zhanyong Wan, Walid Taha, Paul Hudak. Event-Driven FRP. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.

Chapter 16.5

References, Further Reading

Chapter 16: Further Reading (1)

- Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
- Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.
- Conal Elliot. *Functional Implementations of Continuous Modeled Animation*. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.

Chapter 16: Further Reading (2)

- Conal Elliot, Paul Hudak. *Functional Reactive Animation*. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.
- David Harel, Assaf Marron, Gera Weiss. *Behavioral Programming*. Communications of the ACM 55(7):90-100, 2012.
- Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 15, A Module of Reactive Animations; Chapter 18, Higher-Order Types; Chapter 19, An Imperative Robot Language)

Chapter 16: Further Reading (3)

- Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.), *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.
- John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37:67-111, 2000.
- Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.
- Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis. Chalmers University of Technology, 1999.

Chapter 16: Further Reading (4)

- Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001), 229-240, 2001.
- Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 201-222, 2003.
- Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. *Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages*. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.

Chapter 16: Further Reading (5)

- John Peterson, Gregory D. Hager, Paul Hudak. *A Language for Declarative Robotic Programming*. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.
- John Peterson, Paul Hudak, Conal Elliot. *Lambda in Motion: Controlling Robots with Haskell*. In Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), Springer-V., LNCS 1551, 91-105, 1999.
- John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001. `www.haskell.org/frp/manual.html`

Chapter 16: Further Reading (6)

- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 16, Developing reactive functional programs)
- Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. *Prototyping Real-Time Vision Systems: An Experiment in DSL Design*. In Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), 484-493, 1999.
- Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD Thesis, Department of Computer Science, Yale University, December 2002.

Chapter 16: Further Reading (7)

- Zhanyong Wan, Paul Hudak. *Functional Reactive Programming from First Principles*. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), 242-252, 2000.
- Zhanyong Wan, Walid Taha, Paul Hudak. *Real-Time FRP*. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
- Zhanyong Wan, Walid Taha, Paul Hudak. *Event-Driven FRP*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.

Part VI

Extensions, Perspectives

Chapter 17

Extensions to Parallel and ‘Real World’ Functional Programming

Chapter 17.1

Parallelism in Functional Languages

Motivation

...recall:

- Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4):86-90, 2009.

  ...adopting a functional programming style could make your programs more robust, more compact, and **more easily parallelizable.**

Reading for this chapter:

- Peter Pepper, Petra Hofstedt. Funktionale Programmierung, Springer-V., 2006. (In German). (Kapitel 21, Massiv Parallele Programme)

Parallelism in Programming Languages

Predominant in imperative languages:

- Libraries (PVM, MPI) $\rightsquigarrow$ Message Passing Model (C++, C, Fortran)
- Data-parallel Languages (e.g., High Performance Fortran)

Predominant in functional languages:

- Implicit (expression) parallelism
- Explicit parallelism
- Algorithmic skeletons

Implicit Parallelism

...also known as expression parallelism.

Idea: If `f(e1,...,en)` is a functional expression, then

- arguments (and functions) can be evaluated in parallel.

Most important

- advantage: Parallelism for free! No effort for the programmer at all.
- disadvantage: Results often unsatisfying; e.g. granularity, load distribution, etc., is not taken into account.

Overall, expression parallelism is

- easy to detect (for the compiler) but hard to fully exploit.

Explicit Parallelism

Idea: Introducing and using

- meta-statements (e.g., for controlling the data and load distribution, communication).

Most important

- advantage: Often very good results thanks to explicit hands-on control of the programmer.
- disadvantage: High programming effort and loss of functional elegance.

Algorithmic Skeletons

...a compromise between

- explicit imperative parallel programming
- implicit functional expression parallelism

The Setting

...in the following we consider a setting with

- Massively parallel systems
- Algorithmic skeletons

Massively Parallel Systems

...are typically characterized by a

- large number of processors with
  - local memory
  - communication by message exchange
- MIMD-Parallel Processor Architecture (Multiple Instruction/Multiple Data)

Here we focus and restrict ourselves to

- SPMD-Programming Style (Single Program/Multiple Data)

Algorithmic Skeletons

- represent typical patterns for parallelization (Farm, Map, Reduce, Branch&Bound, Divide&Conquer,...).
- are easy to instantiate for the programmer.
- allow parallel programming at a high level of abstraction.

Implementing Algorithmic Skeletons

...in functional languages

- by special higher-order functions.
- with parallel implementation.
- embedded in sequential languages.
- using message passing via skeleton hierarchies.

Advantages:

- Hiding of parallel implementation details in the skeleton.
- Elegance and (parallel) efficiency for special application patterns.

Example: Parallel Map on Distributed List

Consider the higher-order function `map` on lists:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)
```

Observation:

- Applying `f` to a list element does not depend on other list elements.

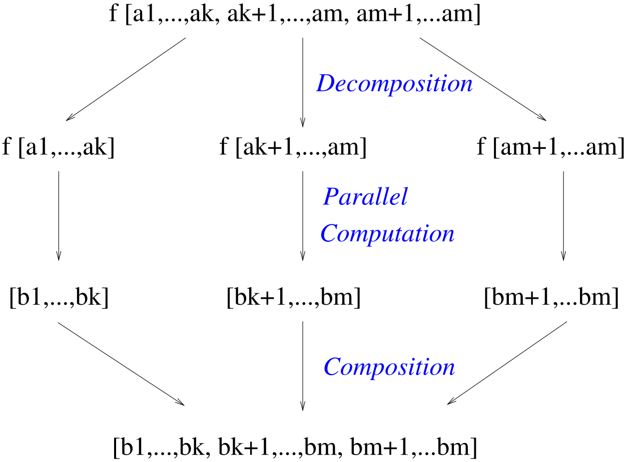
Parallelization idea:

- Divide the list into sublists followed by parallel application of `map` to the sublists:

  $\rightsquigarrow$ parallelization pattern Farm.

Parallel Map on Distributed Lists

Illustration:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung.
Springer, 2006, S. 445.

On the Implementation

Implementing the parallel map function requires

- special data structures, which take into account the aspect of distribution (ordinary lists are inefficient for this purpose).

Skeletons on distributed data structures are so-called

- data-parallel skeletons.

Note the difference between:

- Data-parallelism: Supposes an a priori distribution of data on different processors.
- Task-parallelism: Processes and data to be distributed are not known a priori but dynamically generated.

Programming of a Parallel Application

...using algorithmic skeletons requires:

- Recognizing problem-inherent parallelism.
- Selecting an adequate data distribution (granularity).
- Selecting a suitable skeleton from a library.
- Instantiating the skeleton problem-specifically.

Remark:

- Some languages (e.g., Eden) support the implementation of skeletons (in addition to those which might be provided by a library).

Data Distribution on Processors

...is crucial for

- the structure of the complete algorithm.
- efficiency.

The hardness of the distribution problems depends on

- Independence of all data elements (like in the map-example): Distribution is easy.
- Independence of subsets of data elements.
- Complex dependences of data elements: Adequate distribution is challenging.

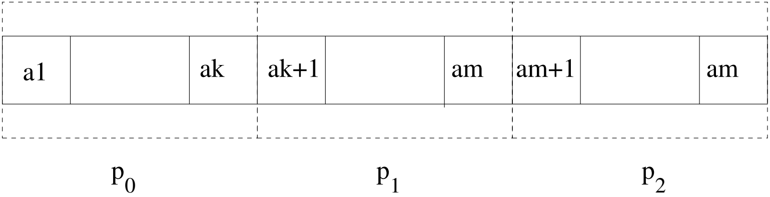
Auxiliary means: So-called covers for

- describing the decomposition and communication pattern of a data structure (investigated by various researchers).

Example (1)

...illlustrating a simple list cover.

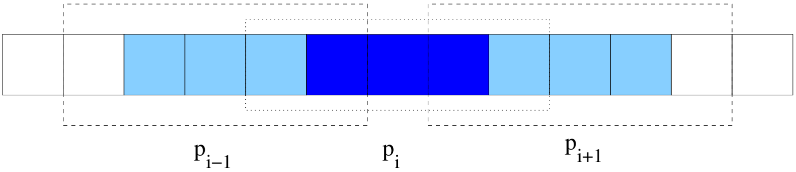
Distributing a list on three processors p_0, p_1, and p_2:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

Example (2)

...illlustrating a list cover with overlapping elements.



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

General Structure

...of a cover:

```
Cover = {
  Type S a           -- Whole object
       C b           -- Cover
       U c           -- Local sub-objects
split :: S a -> C (U a) -- Decomposing the
                        -- original object
glue :: C (U a) -> S a  -- Composing the
                        -- original object
}
```

where it must hold: `glue . split = id`

Note: The above code snippet is not (valid) Haskell.

Implementing Covers

...requires support for

- the specification of covers.
- the programming of algorithmic skeletons on covers.
- the provision of often used skeletons in libraries.

which is currently a

- hot research topic

in functional programming.

Chapter 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 comes mostly in terms of

- sophisticated libraries

and makes Haskell a reasonable choice for addressing and solving

- real world problems

since the choice of a language depends much on the ability and support a programming language provides for linking and connecting to the 'outer world:' the language's

- eco-system.

Chapter 17.3

References, Further Reading

Chapter 17.1: Further Reading (1)

- Joe Armstrong, Robert Virding, Claes Wikstrom, Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.
- Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, Hendrik C.R. Lock. *GOFIN: Higher-Order Functions meet Concurrency Constraints*. Science of Computer Programming 30(1-2):157-199 1998.
- Manuel M.T. Chakravarty, Roman Leshchinsky, Simon Peyton Jones, Gabriele Keller, Simon Marlow. *Data Parallel Haskell: A Status Report*. In Proceedings on the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), ACM, New York, 10-18, 2007.

Chapter 17.1: Further Reading (2)

- Koen Claessen. *A Poor Man's Concurrency Monad*. Journal of Functional Programming 9:313-323, 1999.
- Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 11, Parallel Evaluation)
- Sören Holmström. *PFL: A Functional Language for Parallel Programming*. In Declarative Programming Workshop, 114-139, 1983.
- Peng Li, Simon Marlow, Simon Peyton Jones, Andrew Tolmach. *Lightweight Concurrency Primitives for GHC*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007), 107-118, 2007.

Chapter 17.1: Further Reading (3)

- Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. Higher-Order and Symbolic Computation 16(3):203-251, 2003.
- Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilley, 2013.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 24, Concurrent and Multicore Programming)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 21, Massiv Parallele Programme)

Chapter 17.1: Further Reading (4)

- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 14, Writing parallel functional programs)
- Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*. In Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 295-308, 1996.
- Robert F. Pointon, Philip W. Trinder, Hans-Wolfgang Loidl. *The Design and Implementation of Glasgow Distributed Haskell*. In Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), LNCS 2011, Springer-V., 53-70, 2000.

Chapter 17.1: Further Reading (5)

- Fethi A. Rabhi. *Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach*. In J. R. Davy, P. M. Dew (Eds.), Abstract Machine Models for Highly Parallel Computers, Oxford University Press, 118-139, 1995.
- Fethi A. Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.3, Parallel Algorithms)
- Simon Peyton Jones, Satnam Sing. *A Tutorial on Parallel and Concurrent Programming in Haskell*. *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 267-305, 2008.

Chapter 17.1: Further Reading (6)

- Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming 8(1):23-60, 1998.
- Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Pointon. *Parallel and Distributed Haskells*. Journal of Functional Programming 12(4&5):469-510, 2002.

Chapter 17.2: Further Reading (6)

- Magnus Carlsson, Thomas Hallgren. *Fudgets – A Graphical User Interface in a Lazy Functional Language*. In Proceedings of the 6th ACM International Conference on Functional Programming Languages and Computer Architecture (FPCA'93), 321-330, 1993.
- Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.

Chapter 17.2: Further Reading (7)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 17, Interfacing with C: The FFI; Chapter 19, Error Handling; Chapter 20, Systems Programming in Haskell; Chapter 21, Using Data Bases; Chapter 22, Extended Example: Web Client Programming; Chapter 23, GUI Programming with gtk2hs; Chapter 24, Concurrent and Multicore Programming; Chapter 27, Sockets and Syslog; Chapter 25, Profiling and Optimization; Chapter 28, Software Transactional Memory)
- Thomas Hallgren, Magnus Carlsson. *Programming with Fudgets*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 137-182, 1995.

Chapter 17.2: Further Reading (8)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 19, Agenten und Prozesse; Kapitel 20, Graphische Schnittstellen (GUIs))
- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.
- Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
- "Haskell community." *Hackage: A Repository for Open Source Haskell Libraries*. `hackage.haskell.org`
- "Haskell community." *Haskell wiki*. `haskell.org/haskellwiki/Applications_and_libraries`

Chapter 18

Conclusions and Perspectives

Chapter 18.1

Research Venues, Research Topics, and More

Research Venues, Research Topics, and More

...for functional programming and functional programming languages:

- Research/publication/dissemination venues
  - Conference and Workshop Series
  - Archival Journals
  - Summer Schools
- Research Topics
- Functional Programming in the Real World

Relevant Conference and Workshop Series

For functional programming:

- Annual ACM SIGPLAN International Conference on Functional Programming (ICFP) Series, since 1996.
- Annual Symposium on Functional and Logic Programming (FLPS) Series, since 2000.
- Annual ACM SIGPLAN Haskell Workshop Series, since 2002.
- HAL Workshop Series, since 2006.

For programming in general:

- Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages and Systems (POPL), since 1973.
- Annual ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI), since 1988 (resp. 1973).

Relevant Archival Journals

For functional programming:

- Journal of Functional Programming, since 1991.

For programming in general:

- ACM Transactions on Programming Languages and Systems (TOPLAS), since 1979.
- ACM Computing Surveys, since 1969.

Summer Schools

Focused on functional programming:

- Summer School Series on Advanced Functional Programming. Springer-V., LNCS series.

Hot Research Topics (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."

More on Haskell 2012, Copenhagen, DK, 13 Sep 2012:
http://www.haskell.org/haskell-symposium/2012/

Hot Research Topics (3)

...in theory and practice of functional programming considering the 2012 Call for Papers of ICFP:

"ICFP 2012 seeks original papers on the art and science of functional programming. Submissions are invited on all topics from principles to practice, from foundations to features, and from abstraction to application. The scope includes all languages that encourage functional programming, including both purely applicative and imperative languages, as well as languages with objects, concurrency, or parallelism.

Topics of interest include (but are not limited to):

- Language Design: concurrency and distribution; modules; components and composition; metaprogramming; interoperability; type systems; relations to imperative, object-oriented, or logic programming

Hot Research Topics (4)

- Implementation: abstract machines; virtual machines; interpretation; compilation; compile-time and run-time optimization; memory management; multi-threading; exploiting parallel hardware; interfaces to foreign functions, services, components, or low-level machine resources
- Software-Development Techniques: algorithms and data structures; design patterns; specification; verification; validation; proof assistants; debugging; testing; tracing; profiling
- Foundations: formal semantics; lambda calculus; rewriting; type theory; monads; continuations; control; state; effects; program verification; dependent types
- Analysis and Transformation: control-flow; data-flow; abstract interpretation; partial evaluation; program calculation

Hot Research Topics (5)

- Applications and Domain-Specific Languages: symbolic computing; formal-methods tools; artificial intelligence; systems programming; distributed-systems and web programming; hardware design; databases; XML processing; scientific and numerical computing; graphical user interfaces; multimedia programming; scripting; system administration; security
- Education: teaching introductory programming; parallel programming; mathematical proof; algebra
- Functional Pearls: elegant, instructive, and fun essays on functional programming
- Experience Reports: short papers that provide evidence that functional programming really works or describe obstacles that have kept it from working"

Chapter 18.2

Programming Contest

Programming Contest Series: Background (1)

...considering the 2012 contest edition for illustration.

The ICFP Programming Contest 2012 is the 15th instance of the annual programming contest series sponsored by The ACM SIGPLAN International Conference on Functional Programming. This year, the contest starts at 12:00 July 13 Friday UTC and ends at 12:00 July 16 Monday UTC. There will be a lightning division, ending at 12:00 July 14 Saturday UTC.

The task description will be published at `icfpcontest2012.wordpress.com/task` when the contest starts. Solutions to the task must be submitted online before the contest ends. Details of the submission procedure will be announced along with the contest task.

This is an open contest. Anybody may participate except for the contest organisers and members of the same group as the contest chairs. No advance registration or entry fee is required.

Programming Contest Series: Background (2)

Any programming language(s) may be used as long as the submitted program can be run by the judges on a standard Linux environment with no network connection. Details of the judges' environment will be announced later.

There will be cash prizes for the first and second place teams, the team winning the lightning 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.)...

More on ICFP 2012, Copenhagen, DK, 10-12 Sep 2012:
http://icfpconference.org/icfp2012/cfp.html

The 22nd Programming Contest at ICFP 2019

In 2019, the programming contest will start on

- Friday 21 June 2019 10:00am UTC. The 24hr lightning division will end at Saturday 22 June 2019 10:00am UTC and the 72hr full contest will end at Monday 24 June 2019 10:00am UTC; full information is available online:

  `https://icfpcontest2019.github.io`

- Stay tuned for news on the
  - programming contest series at the ICFP conf. series:
    `https://www.icfpconference.org/contest.html`
  - 22nd programming contest edition in 2019:
    `https://icfpcontest2019.github.io/`
  - 2019 host conference:
    ICFP 2019, Berlin, Germany, August 18-23, 2019:
    `https://icfp19.sigplan.org/home`

Chapter 18.3

In Conclusion

Functional Programming

...certainly arrived in the real world:

- Curt J. Simpson. Experience Report: Haskell in the “Real World”: Writing a Commercial Application in a Lazy Functional Language. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
- Jerzy Karczmarczuk. Scientific Computation and Functional Programming. Computing in Science and Engineering 1(3):64-72, 1999.
- Bryan O'Sullivan, John Goerzen, Don Stewart. Real World Haskell. O'Reilly, 2008.
- Haskell in Industry and Open Source: `www.haskell.org/haskellwiki/Haskell_in_industry`

A Plea for Functional Programming

...even though

- Philip Wadler. Why no one uses Functional Languages. ACM SIGPLAN Notices 33(8):23-27, 1998.
- Philip Wadler. An angry half-dozen. ACM SIGPLAN Notices 33(2):25-30, 1998.

might suggest the opposite, which, however, is actually not true, and Philip Wadler's apparent lamentation is more an impassioned

- plea for functional programming

in the real world summarizing a number of very general obstacles preventing good or even superior ideas also in the field of programming to make their way into mainstream practices easily and fast.

More Pleas for Functional Programming

...in the real world:

- Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4): 86-90, 2009.
- Konstantin Läufer, Geoge K. Thiruvathukal. The Promises of Typed, Pure, and Lazy Functional Programming: Part II. Computing in Science and Engineering 11(5): 68-75, 2009.
- Yaron Minsky. OCaml for the Masses. Communications of the ACM, 54(11):53-58, 2011.

and brand-new:

- Neil Savage. Using Functions for Easier Programming. Communications of the ACM 61(5):29-30, 2018.

Recall Edsger W. Dijkstra's Prediction

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.(∗)

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

(∗) Quote from: Introducing a course on calculi. Announcement of a lecture course at the University of Texas at Austin, 1995.

In the Words of Simon Peyton Jones

When the limestone of imperative programming has worn away,
the granite of functional programming
will be revealed underneath.

Simon Peyton Jones

In the Words of John Carmack

Sometimes, the elegant implementation is a function.
Not a method. Not a class. Not a framework.
Just a function.

John Carmack

Chapter 18.4

References, Further Reading

Chapter 18: Further Reading (1)

- Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.
- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.
- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC.
- Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.

Chapter 18: Further Reading (2)

- Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. Journal of Functional Programming 7(2):163-182, 1997.
- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.
- Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.
- Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.

Chapter 18: Further Reading (3)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 25, Profiling and Optimization)
- Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
- Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
- David A. Turner. *Total Functional Programming*. Journal of Universal Computer Science 10(7):751-768, 2004.

Chapter 18: Further Reading (4)

- Marcos Viera, S. Doaitse Swierstra, Wouter S. Swierstra. *Attribute Grammars fly First Class: How do we do Aspect Oriented Programming in Haskell*. In Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009), 245-256, 2009.
- Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2):25-30, 1998.
- Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8):23-27, 1998.
- "Haskell community." *Haskell in Industry and Open Source*. www.haskell.org/haskellwiki/Haskell_in_industry

References

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)

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- Martin Aigner, Günter M. Ziegler. *Proofs from the Book*. Springer-V., 4th edition, 2010.
- Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
- Joe Armstrong, Robert Virding, Claes Wikstrom, Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.
- André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.

I Textbooks (2)

- Manoochehr Azmoodeh. *Abstract Data Types and Algorithms*. Macmillan Education, 1988.
- Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised edition, North Holland, 1984.
- Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- Richard E. Bellman, Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1957.
- Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012.

I Textbooks (3)

- Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013.
- Richard Bird. *Introduction to Functional Programming using Haskell*. 2nd edition, Prentice Hall, 1998.
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015.
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
- William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
- Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

I Textbooks (4)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011.
- Manuel M.T. Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004.
- Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. 2nd edition, MIT Press, 2001.
- Peter Crawley, Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.

I Textbooks (5)

- H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. `citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf`
- Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.
- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012.

I Textbooks (6)

- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004.
- Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.
- Marcel Erné. *Einführung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999.
- Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

I Textbooks (7)

- Helmuth Gericke. *Theorie der Verbände*. Bibliographisches Institut, 2. Auflage, 1967.
- George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 1998.
- George Grätzer. *Lattice Theory: Foundation*. Birkhäuser, 2011.
- Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions – An Introduction to Computer Science using Scheme*. Brooks/Cole Publishing Company, 1999.
- Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001.

I Textbooks (8)

- Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004.
- Rachel Harrison. *Abstract Data Types in Standard ML*. J. Wiley, 1993.
- Peter Henderson. *Functional Programming: Application and Implementation*. Prentice Hall, 1980.
- Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

I Textbooks (9)

- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.
- Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009.
- Mark P. Jones, Alastair Reid et al. (Eds.). *The Hugs98 User Manual*. `www.haskell.org/hugs`
- Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1952. (Reprint, North Holland, 1980)
- Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006.
- Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.

I Textbooks (10)

- Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009.
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011.
- Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998.
- Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
- David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008.
- Kimbal Marriott, Peter J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

I Textbooks (11)

- Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilley, 2013.
- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2nd edition, 2011.
- Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005.

I Textbooks (12)

- Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- John O'Donnell, Cordelia Hall, Rex Page. *Discrete Mathematics Using a Computer*. Springer-V., 2nd edition, 2006.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008.
- Lawrence C. Paulson. *Logic and Computation – Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003.

I Textbooks (13)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006.
- Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
- Simon Peyton Jones, David Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.
- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.
- Fethi A. Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999.

I Textbooks (14)

- Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
- George E. Revesz. *Lambda-Calculus, Combinators and Functional Programming*. Cambridge University Press, 1988.
- Steven Roman. *Lattices and Ordered Sets*. Springer-V., 2008.
- Gunter Saake, Kai-Uwe Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 4. überarbeitete Auflage, 2010.
- Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

I Textbooks (15)

- Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002.
- Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998.
- Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik: Induktives Vorgehen*. Springer-V., 2014.
- Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018.
- Aaron Stump. *Verified Functional Programming in Agda*. ACM Books Series, No. 9, 2016.

I Textbooks (16)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011.
- Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.
- Daniel J. Velleman. *How to Prove It. A Structured Approach*. Cambridge University Press, 1994.
- Mitchell Wand. *Induction, Recursion, and Programming*. Elsevier, 1980.

II Monographs (1)

- Jon R. Bentley. *Programming Pearls*. Addison-Wesley, 1987.
- Jon R. Bentley. *Programming Pearls*. Addison-Wesley, 2nd edition, 2000. (Excerpt of the book online available from `www.cs.bell-labs.com/cm/cs/pearls`)
- Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011.
- Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.

II Monographs (2)

- Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis. University of Cambridge, British Computer Society Distinguished Dissertations in Computer Science, Cambridge University Press, 1992.
- Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-V., 1971 (2nd edition, 1998).
- Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis. Chalmers University of Technology, 1999.
- Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD thesis. Department of Computer Science, Yale University, December 2002.

III Volumes (1)

- Roland Backhouse, Roy Crole, Jeremy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002.
- Jeremy Gibbons, Oege de Moor (Eds.). *The Fun of Programming*. Palgrave MacMillan, 2003.
- George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. I*. Birkhäuser, 2014.
- George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. II*. Birkhäuser, 2016.

III Volumes (2)

- Johan Jeuring, Erik Meijer (Eds.). *Advanced Functional Programming*. Springer-V., LNCS 925, 1995.
- Johan Jeuring, Simon Peyton Jones (Eds.). *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 2003.
- Pieter Koopman, Rinus Plasmeijer, S. Doaitse Swierstra (Eds.). *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 2008.
- Peter Rechenberg, Gustav Pomberger (Eds.). *Informatik-Handbuch*. Carl Hanser Verlag, 4th edition, 2006.
- Colin Runciman, David Wakeling (Eds.). *Applications of Functional Programming*. UCL Press, 1995.

III Volumes (3)

- Davide Sangiorgi, Jan Rutten (Eds.). *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Vol. 52, Cambridge University Press, 2011.
- S. Doaitse Swierstra, Pedro Rangel Henriques, José Nuno Oliveira (Eds.). *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 1608, 1999.

IV Articles (1)

- Umut A. Acar, Guy E. Blelloch, Robert Harper. *Selective Memoization*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 14-25, 2003.
- Hassan Ait-Kaci, Roger Nasr. *Integrating Logic and Functional Programming*. Lisp and Symbolic Computation 2(1):51-89, 1989.
- Sergio Antoy, Michael Hanus. *Compiling Multi-Paradigm Declarative Languages into Prolog*. In Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2000), Springer-V., LNCS 1794, 171-185, 2000.
- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.

IV Articles (2)

- Sergio Antoy, Michael Hanus. *New Functional Logic Design Patterns*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 19-34, 2011.
- Henry G. Baker. *Shallow Binding Makes Functional Arrays Fast*. ACM SIGPLAN Notices 26(8):145-147, 1991.
- Falk Bartels. *Generalized Coinduction*. Electronic Notes in Theoretical Computer Science 44:1, 21 pages, 2001. http://www.elsevier.nl/locate/entcs/volume44.html
- Falk Bartels. *Generalized Coinduction*. Journal of Mathematical Structures in Computer Science 13(2):321-348, 2003.
- Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.

IV Articles (3)

- Richard Bird. *Fifteen Years of Functional Pearls*. In Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 215, 2006.
- Richard Bird. *How to Write a Functional Pearl*. Invited presentation at the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 2006. http://icfp06.cs.uchicago.edu/bird-talk.pdf
- Garret Birkhoff. *Applications of Lattice Algebra*. Mathematical Proceedings of the Cambridge Philosophical Society 30(2):115-122, 1934.
- James R. Bitner, Edward M. Reingold. *Backtrack Programming Techniques*. Communications of the ACM 18(11):651-656, 1975.

IV Articles (4)

- Matthias Blume, David McAllester. *Sound and Complete Models of Contracts*. Journal of Functional Programming 16(4-5):375-414, 2006.
- Ana Bove, Peter Dybjer, Andrés Sicard-Ramírez. *Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs*. In Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2012), Springer-V., LNCS 7213, 104-118, 2012.
- Bernd Braßel, Michael Hanus, Björn Peemöller, Fabian Reck. *KiCS2: A New Compiler from Curry to Haskell*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 1-18, 2011.

IV Articles (5)

- F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.
- Magnus Carlsson, Thomas Hallgren. *Fudgets – A Graphical User Interface in a Lazy Functional Language*. In Proceedings of the 6th ACM International Conference on Functional Programming Languages and Computer Architecture (FPCA'93), 321-330, 1993.
- Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, Hendrik C.R. Lock. *GOFIN: Higher-Order Functions meet Concurrency Constraints*. Science of Computer Programming 30(1-2):157-199 1998.

IV Articles (6)

- Manuel M.T. Chakravarty, Gabriele Keller. *An Approach to Fast Arrays in Haskell*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 27-58, 2003.
- Manuel M.T. Chakravarty, Roman Leshchinsky, Simon Peyton Jones, Gabriele Keller, Simon Marlow. *Data Parallel Haskell: A Status Report*. In Proceedings on the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), ACM, New York, 10-18, 2007.
- Stephen Chang, Matthias Felleisen. *The Call-by-Need Lambda Calculus, Revisited*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 128-147, 2012.

IV Articles (7)

Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.

Olaf Chitil. *Pretty Printing with Lazy Dequeues*. In Proceedings of the ACM SIGPLAN 2001 Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.

Jan Christiansen, Sebastian Fischer. *Easycheck – Test Data for Free*. In Proceedings of the 9th International Symposium on Functional and Logic Programming (FLPS 2008), Springer-V., LNCS 4989, 322-336, 2008.

Koen Claessen. *A Poor Man's Concurrency Monad*. Journal of Functional Programming 9:313-323, 1999.

IV Articles (8)

- Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings ACM of the SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.
- Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 17-39, 2003.

IV Articles (9)

- Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, Malcolm Wallace. *Testing and Tracing Lazy Functional Programs Using `Quickcheck` and `Hat`*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 59-99, 2003.
- Byron Cook, John Launchbury. *Disposable Memo Functions*. In Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97), 310, 1997 (full paper in Proceedings Haskell'97 workshop).
- Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.

IV Articles (10)

- Werner Damm, Bernhard Josko. *A Sound and Relatively* Complete Hoare-Logic for a Language with Higher Type Procedures*. Acta Informatica 20:59-101, 1983.
- Anne C. Davis. *A Characterization of Complete Lattices*. Pacific Journal of Mathematics 5(2):311-319, 1955.
- Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98). Springer-V., LNCS 1422, 122-150, 1998.
- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.

IV Articles (11)

- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC
- Norbert Eisinger, Tim Geisler, Sven Panne. *Logic Implemented Functionally*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-V., LNCS 1292, 351-368, 1997.
- Conal Elliot. *Functional Implementations of Continuous Modeled Animation*. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.

IV Articles (12)

- Conal Elliot, Paul Hudak. *Functional Reactive Animation*. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.
- Kento Emoto, Sebastian Fischer, Zhenjiang Hu. *Generate, Test, and Aggregate: A Calculation-based Framework for Systematic Parallel Programming with MapReduce*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 254-273, 2012.
- Jeroen Fokker. *Functional Parsers*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 1-23, 1995.

IV Articles (13)

- Daniel P. Friedman, David S. Wise. *CONS should not Evaluate its Arguments*. In Proceedings of the 3rd International Conference on Automata, Languages and Programming, 257-284, 1976.
- Jeremy Gibbons. *Functional Pearls – An Editor's Perspective*. www.cs.ox.ac.uk/people/jeremy.gibbons/pearls/
- Andy Gill, Simon Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995. `www.haskell.org/happy`
- Andreas Goerdt. *A Hoare Calculus for Functions defined by Recursion on Higher Types*. In Proceedings of the Conference on Logic of Programs, Springer-V, LNCS 193, 106-117, 1985.

IV Articles (14)

- David Gries. *The Maximum Segment Sum Problem*. In *Formal Development of Programs and Proofs*. Edsger W. Dijkstra (Ed.), Addison-Wesley (UT Year of Programming Series), 43-45, 1990.
- Klaus E. Grue. *Arrays in Pure Functional Programming Languages*. International Journal on Lisp and Symbolic Computation 2:105-113, Kluwer Academic Publishers, 1989.
- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
- John V. Guttag, James J. Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.

IV Articles (15)

- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.
- Anthony Hall, Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. IEEE Software 19(1):18-25, 2002.
- Thomas Hallgren, Magnus Carlsson. *Programming with Fudgets*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 137-182, 1995.
- Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994.

IV Articles (16)

- Michael Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice*. Journal of Functional Programming 19&20:583-628, 1994.
- Michael Hanus (Ed.). *Curry: An Integrated Functional Logic Language*. Vers. 0.8.2, 2006. `www.curry-language.org/`
- Michael Hanus. *Multi-paradigm Declarative Languages*. In Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Springer-V., LNCS 4670, 45-75, 2007.
- Michael Hanus. *Functional Logic Programming: From Theory to Curry*. In *Programming Logics – Essays in Memory of Harald Ganzinger*. Springer-V., LNCS 7797, 123-168, 2013.

IV Articles (17)

- Michael Hanus, Sergio Antoy, Bernd Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner. *PAKCS: The Portland Aachen Kiel Curry System*. 2013. Available at `www.informatik.uni-kiel.de/~pakcs`

- Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. *Curry: A Truly Functional Logic Language*. In Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming, 95-107, 1995.

- David Harel, Assaf Marron, Gera Weiss. *Behavioral Programming*. Communications of the ACM 55(7):90-100, 2012.

- Peter Henderson, James H. Morris. *A Lazy Evaluator*. In Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages (POPL'76), 95-103, 1976.

IV Articles (18)

- Steve Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6(3):445-464, 1996.
- Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
- Charles A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM 12(10):576-580, 1969.
- Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.
- Charles A.R. Hoare. *Retrospective: An Axiomatic Basis for Computer Programming*. Communications of the ACM 52(10):30-32, 2009.

IV Articles (19)

- Sören Holmström. *PFL: A Functional Language for Parallel Programming*. In Declarative Programming Workshop, 114-139, 1983.
- Paul Hudak. *Arrays, Non-determinism, Side-effects, and Parallelism: A Functional Perspective*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 312-327, 1986.
- Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.

IV Articles (20)

- John Hughes. *Lazy Memo Functions*. In Proceedings of the IFIP Symposium on Functional Programming Languages and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 129-146, 1985.
- John Hughes. *An Efficient Implementation of Purely Functional Arrays*. Technical Report, Programming Methodology Group, Chalmers University of Technology, 1985.
- John Hughes. *Why Functional Programming Matters*. Computer Journal 32(2):98-107, 1989.
- John Hughes. *The Design of a Pretty-Printer Library*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 53-96, 1995.

IV Articles (21)

- John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37:67-111, 2000.
- Chung-Kil Hur, Georg Neis, Derek Dreyer, Viktor Vafeiadis. *The Power of Parameterization in Coinductive Proofs*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 193-205, 2013.
- Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
- Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
- Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 8(4):437-444, 1998.

IV Articles (22)

- Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. Journal of Functional Programming 7(2):163-182, 1997.
- Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62:222-259, 1997.
- Joxan Jaffar, Jean-Louis Lassez. *Constraint Logic Programming*. In Conference Record of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'87), 111-119, 1987.

IV Articles (23)

- Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko. *HMC: Verifying Functional Programs using Abstract Interpreters*. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), Springer-V., LNCS 6806, 470-485, 2011.
- Mark P. Jones. *Functional Thinking*. Lecture at the 6th International Summer School on Advanced Functional Programming, Boxmeer, The Netherlands, 2008.
- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.

IV Articles (24)

- Naoki Kobayashi, Ryosuke Sato, Hiroshi Unno. *Predicate Abstraction and CEGAR for Higher-Order Model Checking*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), 222-233, 2011.
- Pieter W.M. Koopman, Marinus J. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), Selected Papers, Springer-V., LNCS 1595, 120-136, 1999.
- Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.

IV Articles (25)

- Peter J. Landin. *A Correspondence between ALGOL60 and Church's Lambda-Notation: Part I*. Communications of the ACM 8(2):89-101, 1965.
- Guy Lapalme, Fabrice Lavier. *Using a Functional Language for Parsing and Semantic Processing*. Computational Intelligence 9(2):111-131, 1993.
- John Launchbury, Simon Peyton Jones. *State in Haskell*. Lisp and Symbolic Computation 8(4):293-341, 1995.
- Daan Leijen. *Parsec, a free Monadic Parser Combinator Library for Haskell*, 2003. `legacy.cs.uu.nl/daan/parsec.html`
- Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. Electronic Notes in Theoretical Computer Science 41(1), 20 pages, 2001.

IV Articles (26)

- Marina Lenisa. *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*. Electronic Notes in Theoretical Computer Science 19:2-22, 1999.
- Peng Li, Simon Marlow, Simon Peyton Jones, Andrew Tolmach. *Lightweight Concurrency Primitives for GHC*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007), 107-118, 2007.
- John W. Lloyd. *Programming in an Integrated Functional and Logic Language*. Journal of Functional and Logic Programming 1999(3), 49 pages, MIT Press, 1999.
- Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. Higher-Order and Symbolic Computation 16(3):203-251, 2003.

IV Articles (27)

Rita Loogen, Yolanda Ortega-Mallén, Ricardo Pena-Mari. *Parallel Functional Programming in Eden*. Journal of Functional Programming 15(3):431-475, 2005.

Francisco J. López-Fraguas, Jaime Sánchez-Hernández. *TOY: A Multi-paradigm Declarative System*. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), Springer-V., LNCS 1631, 244-247, 1999.

George Markowsky. *Chain-complete Posets and Directed Sets with Applications*. Algebra Universalis 6(1):53-68, 1976.

Lambert Meertens. *Functional Pearl: Calculating the Sieve of Eratosthenes*. Journal of Functional Programming 14(6):759-763, 2004.

IV Articles (28)

- Donald Michie. *'Memo' Functions and Machine Learning*. Nature, 218:19-22, 1968.
- Matthew Might, David Darais, Daniel Spiewak. *Parsing with Derivatives – A Functional Pearl*. In Proceedings of the 16th ACM International Conference on Functional Programming (ICFP 2011), 189-195, 2011.
- Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
- Neil Mitchell, Colin Runciman. *Not all Patterns, but enough: An Automated Verifier for Partial but Succifient Pattern Matching*. In Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell 2008), 49-60, 2008.

IV Articles (29)

- Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
- Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
- Juan Jose Moreno-Navarro, Mario Rodriguez-Artalejo. *Logic Programming with Functions and Predicates: The Language BABEL*. Journal of Logic Programming 1(3-4):191-223, 1992.
- Shin-Cheng Mu. *The Maximum Segment Sum is Back: Deriving Algorithms for two Segment Problems with Bounded Lengths*. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2008), 31-39, 2008.

IV Articles (30)

- Martin Müller, Tobias Müller, Peter Van Roy. *Multiparadigm Programming in Oz*. In Proceedings of the Workshop on Visions for the Future of Logic Programming (ILPS'95), 1995.
- Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.
- Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.

IV Articles (31)

- Chris Okasaki. *Simple and Efficient Purely Functional Queues and Dequeues*. Journal of Functional Programming 5(4):583-592, 1995.
- Matti Nykänen. *A Note on the Genuine Sieve of Eratosthenes*. Journal of Functional Programming 21(6):563-572, 2011.
- Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. Journal of Functional Programming 19(1):95-106, 2009.
- Melissa E. O'Neill, F. Warren Burton. *A New Method for Functional Arrays*. Journal of Functional Languages 7(5):487-513, 1997.
- Derek Oppen. *Pretty-printing*. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.

IV Articles (32)

- Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001), 229-240, 2001.
- Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 201-222, 2003.
- Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. *Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages*. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.

IV Articles (33)

- John Peterson, Gregory D. Hager, Paul Hudak. *A Language for Declarative Robotic Programming*. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.
- John Peterson, Paul Hudak, Conal Elliot. *Lambda in Motion: Controlling Robots with Haskell*. In Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), Springer-V., LNCS 1551, 91-105, 1999.
- John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001. `www.haskell.org/frp/manual.html`

IV Articles (34)

- Thomas Petricek. *What We Talk about when We Talk about Monads*. The Art, Science, and Engineering of Programming 2(3), Article 12, 1-27, 2018.
- Simon Peyton Jones. *Haskell pretty-printer library*. 1997. `www.haskell.org/libraries/#prettyprinting`.
- Simon Peyton Jones. *Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell*. In Tony Hoare, Manfred Broy, Ralf Steinbruggen (Eds.), Engineering Theories of Software Construction, IOS Press, 47-96, 2001 (Presented at the 2000 Marktoberdorf Summer School).
- Simon Peyton Jones. *Haskell 98 Libraries: Arrays*. Journal of Functional Programming 13(1):173-178, 2003.

IV Articles (35)

- Simon Peyton Jones, Jean-Marc Eber, Julian Seward. *Composing Contracts: An Adventure in Financial Engineering*. In Proceedings of the 5th ACM SIGPLAN Conference on Functional Programming (ICFP 2000), 280-292, 2000.
- Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*. In Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 295-308, 1996.
- Simon Peyton Jones, John Launchbury. *State in Haskell*. Lisp and Symbolic Computation 8(4):293-341, 1995.

IV Articles (36)

- Simon Peyton Jones, Satnam Sing. *A Tutorial on Parallel and Concurrent Programming in Haskell*. *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 267-305, 2008.
- Simon Peyton Jones, Philip Wadler. *Imperative Functional Programming*. In Conference Record of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 71-84, 1993.
- Robert F. Pointon, Philip W. Trinder, Hans-Wolfgang Loidl. *The Design and Implementation of Glasgow Distributed Haskell*. In Proceedings of the 12th 97International Workshop on Implementation of Functional Languages (IFL 2000), Springer-V., LNCS 2011, 53-70, 2000.

IV Articles (37)

- Fethi A. Rabhi. *Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach*. In J. R. Davy, P. M. Dew (Eds.), Abstract Machine Models for Highly Parallel Computers, Oxford University Press, 118-139, 1995.
- Uday S. Reddy. *Narrowing as the Operational Semantics of Functional Languages*. In Proceedings of the IEEE International Symposium on Logic Programming, 138-151, 1985.
- Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. *Prototyping Real-Time Vision Systems: An Experiment in DSL Design*. In Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), 484-493, 1999.

IV Articles (38)

- Tillmann Rendel, Klaus Ostermann. *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing*. In Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell 2010), 1-12, 2010.
- Colin Runciman. *Lazy Wheel Sieves and Spirals of Primes*. Journal of Functional Programming 7(2):219-225, 1997.
- Colin Runciman, Matthew Naylor, Fredrik Lindblad. *Small-Check and Lazy SmallCheck*. In Proceedings of the ACM SIGPLAN 2008 Workshop on Haskell (Haskell 2008), 37-48, 2008. `http://hackage.haskell.org`

IV Articles (39)

- Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. Theoretical Computer Science 308:1-53, 2003.
- Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*. Susan Eisenbach (Ed.), Ellis Horwood, 7-8, 1987.
- Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.

IV Articles (40)

- Tom Schrijvers, Peter J. Stuckey, Philip Wadler. *Monadic Constraint Programming*. Journal of Functional Programming 19(6):663-697, 2009.
- Silvija Seres, Michael Spivey. *Embedding Prolog in Haskell*. In Proceedings of the 1999 Haskell Workshop (Haskell'99), Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, 25-38, 1999.
- Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

IV Articles (41)

- Zoltan Somogyi, Fergus Henderson, Thomas Conway. *The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language*. Journal of Logic Programming 29(1-3):17-64, 1996.
- Zoltan Somogyi, Fergus J. Henderson, Thomas C. Conway. *Mercury: An Efficient Purely Declarative Logic Programming Language*. In Proceedings of the 18th Australasian Computer Science Conference, 499-512, 1995.
- William Sonnex, Sophia Drossopoulou, Susan Eisenbach. *Zeno: An Automated Prover for Properties of Recursive Data Structures*. In Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012), Springer-V., LNCS 7214, 407-421, 2012.

IV Articles (42)

- Jay M. Spitzen, Karl M. Levitt, Lawrence Robinson. *An Example of Hierarchical Design and Proof*. Communications of the ACM 21(12):1064-1075, 1978.
- Michael Spivey. *A Functional Theory of Exceptions*. Science of Computer Programming 14(1):25-42, 1990.
- Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 177-199, 2003.

IV Articles (43)

- Philippe Suter, Ali Sinan Köksal, Viktor Kuncak. *Satisfiability Modulo Recursive Programs*. In Proceedings of the 18th International Conference on Static Analysis (SAS 2011), Springer-V., LNCS 6887, 298-315, 2011.
- S. Doaitse Swierstra. *Combinator Parsing: A Short Tutorial*. In Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Revised Tutorial Lectures. Springer-V., LNCS 5520, 252-300, 2009.
- S. Doaitse Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer-V., LNCS 1725, 111-129, 1999.

IV Articles (44)

- S. Doaitse Swierstra, Luc Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer-V., LNCS 1129, 184-207, 1996.
- Wouter S. Swierstra, Thorsten Altenkirch. *Beauty in the Beast: A Functional Semantics for the Awkward Sqad*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007), 25-36, 2007.
- Alfred Tarski. *A Lattice-theoretical Fixpoint Theorem and its Applications*. Pacific Journal of Mathematics 5(2):285-309, 1955.
- Simon Thompson. *Proof*. In *Research Directions in Parallel Functional Programming*, Kevin Hammond, Greg Michaelson (Eds.), Springer-V., Chapter 4, 93-119, 1999.

IV Articles (45)

- Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming 8(1):23-60, 1998.
- Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Pointon. *Parallel and Distributed Haskells*. Journal of Functional Programming 12(4&5):469-510, 2002.
- David A. Turner. *Total Functional Programming*. Journal of Universal Computer Science 10(7):751-768, 2004.
- Hiroshi Unno, Tachio Terauchi, Naoki Kobayashi. *Automating Relatively Complete Verification of Higher-Order Functional Programs*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 75-86, 2013.

IV Articles (46)

- Marcos Viera, S. Doaitse Swierstra, Wouter S. Swierstra. *Attribute Grammars fly First Class: How do we do Aspect Oriented Programming in Haskell*. In Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009), 245-256, 2009.
- Dimitrios Vytiniotis, Simon Peyton Jones, Dan Rosén, Koen Claessen. *HALO: Haskell to Logic through Denotational Semantics*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 431-442, 2013.
- Philip Wadler. *How to Replace Failure with a List of Successes*. In Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 113-128, 1985.

IV Articles (47)

- Philip Wadler. *A New Array Operation*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 328-335, 1986.
- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.
- Philip Wadler. *Monads for Functional Programming*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 24-52, 1995.
- Philip Wadler. *How to Declare an Imperative*. In Proceedings of the 1995 International Symposium on Logic Programming (ILPS'95), Invited Presentation, MIT Press, 18-32, 1995.

IV Articles (48)

- Philip Wadler. *How to Declare an Imperative*. ACM Computing Surveys 29(3):240-263, 1997.
- Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2): 25-30, 1998.
- Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8): 23-27, 1998.
- Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 223-243, 2003.

IV Articles (49)

- Zhanyong Wan, Paul Hudak. *Functional Reactive Programming from First Principles*. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), 242-252, 2000.
- Zhanyong Wan, Walid Taha, Paul Hudak. *Real-Time FRP*. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
- Zhanyong Wan, Walid Taha, Paul Hudak. *Event-Driven FRP*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.

V Haskell 98 – Language Definition

- Paul Hudak, Simon Peyton Jones, Philip Wadler (Eds.). *Report on the Programming Language Haskell: Version 1.1*. Technical Report, Yale University and Glasgow University, August 1991.
- Paul Hudak, Simon Peyton Jones, Philip Wadler (Eds.). *Report on the Programming Language Haskell: A Non-strict Purely Funcional Language (Version 1.2)*. ACM SIGPLAN Notices 27(5):1-164, 1992.
- Simon Marlow (Ed.). *Haskell 2010 Language Report*, 2010. `www.haskell.org/definition/haskell2010.pdf`
- Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. `www.haskell.org/definitions`

VI The History of Haskell

- Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the Occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL 2003), 2003. `research.microsoft.com/users/simonpj/papers/haskell-retrospective/`
- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007 (ACM Digital Library `www.acm.org/dl`)

Appendix

Appendix A

Mathematical Foundations

A.1

Relations

Relations

Let M_i, $1 \leq i \leq k$, be sets.

Definition A.1.1 (k-ary Relation)

A (k-ary) relation is a set R of ordered tuples of elements of $M_1, \ldots, M_k$, i.e., $R \subseteq M_1 \times \ldots \times M_k$ is a subset of the cartesian product of the sets M_i, $1 \leq i \leq k$.

Examples

- $\emptyset$ is the smallest relation on $M_1 \times \ldots \times M_k$.
- $M_1 \times \ldots \times M_k$ is the biggest relation on $M_1 \times \ldots \times M_k$.

Binary Relations

Let M, N be sets.

Definition A.1.2 (Binary Relation)

A (binary) relation is a set R of ordered pairs of elements of M and N, i.e., R is a subset of the cartesian product of M and N, $R \subseteq M \times N$, called a relation from M to N.

Examples

- $\emptyset$ is the smallest relation from M to N.
- $M \times N$ is the biggest relation from M to N.

Note

- If R is a relation from M to N, it is common to write $m\,R\,n$, $R(m,n)$, or $R\,m\,n$ instead of $(m,n) \in R$.

Between, On

Definition A.1.3 (Between, On)

A relation R from M to N is called a relation between M and N (or a relation on $M \times N$).

If M equals N, then R is called a relation on M, in symbols: (M, R).

Domain and Range of a Binary Relation

Definition A.1.4 (Domain and Range)

Let R be a relation from M to N.

The sets

- $dom(R) =_{df} \{m \mid \exists\, n \in N.\ (m, n) \in R\}$
- $ran(R) =_{df} \{n \mid \exists\, m \in M.\ (m, n) \in R\}$

are called the domain and the range of R, respectively.

Properties of Relations on a Set M

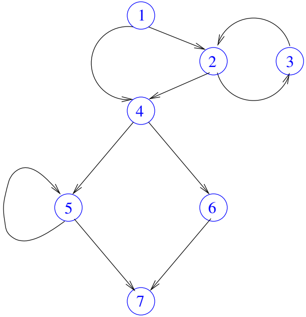
Definition A.1.5 (Properties of Relations on M)

A relation R on a set M is called

- reflexive iff $\forall\, m \in M.\ m\, R\, m$
- irreflexive iff $\forall\, m \in M.\ \neg m\, R\, m$
- transitive iff $\forall\, m, n, p \in M.\ m\, R\, n\ \wedge\ n\, R\, p\ \Rightarrow\ m\, R\, p$
- intransitive iff $\forall\, m, n, p \in M.\ m\, R\, n \wedge n\, R\, p \Rightarrow \neg m\, R\, p$
- symmetric iff $\forall\, m, n \in M.\ m\, R\, n\ \Longleftrightarrow\ n\, R\, m$
- antisymmetric iff $\forall\, m, n \in M.\ m\, R\, n \wedge n\, R\, m \Rightarrow\ m = n$
- asymmetric iff $\forall\, m, n \in M.\ m\, R\, n \Rightarrow \neg n\, R\, m$
- linear iff $\forall\, m, n \in M.\ m\, R\, n\ \vee n\, R\, m\ \vee\ m = n$
- total iff $\forall\, m, n \in M.\ m\, R\, n\ \vee\ n\, R\, m$

(Anti-) Example

Let $G = (N, E, \mathbf{s} \equiv 1, \mathbf{e} \equiv 7)$ be the below (flow) graph, and let R be the relation '$\cdot$ is linked to $\cdot$ via a (directed) edge' on N of G (e.g., node 4 is linked to node 6 but not vice versa).



The relation R is not reflexive, not irreflexive, not transitive, not intransive, not symmetric, not antisymmetric, not asymmetric, not linear, and not total.

Equivalence Relation

Let R be a relation on M.

Definition A.1.6 (Equivalence Relation)

R is an equivalence relation (or equivalence) iff R is reflexive, transitive, and symmetric.

Exercise A.1.7

Let $|$ denote the divisibility relation on the set of natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \mid 35$.

Prove or disprove: The divisibility relation $|$ on $\mathbb{N}_0$ is

1. reflexive
2. irreflexive
3. transitive
4. intransitive
5. symmetric
6. antisymmetric
7. asymmetric
8. linear
9. total
10. equivalence (relation)

Proof or counterexample.

A.2

Ordered Sets

A.2.1

Pre-Orders, Partial Orders, and More

Ordered Sets

Let R be a relation on M.

Definition A.2.1.1 (Pre-Order)

R is a pre-order (or quasi-order) iff R is reflexive and transitive.

Definition A.2.1.2 (Partial Order)

R is a partial order (or poset or order) iff R is reflexive, transitive, and antisymmetric.

Definition A.2.1.3 (Strict Partial Order)

R is a strict partial order iff R is asymmetric and transitive.

Examples of Ordered Sets

Pre-order (reflexive, transitive)

- The relation $\Rightarrow$ on logical formulas.

Partial order (reflexive, transitive, antisymmetric)

- The relations $=$, $\leq$ and $\geq$ on $\mathbb{N}$.
- The relation $m \mid n$ (m is a divisor of n) on $\mathbb{N}$.

Strict partial order (asymmetric, transitive)

- The relations $<$ and $>$ on $\mathbb{N}$.
- The relations $\subset$ and $\supset$ on sets.

Equivalence relation (reflexive, transitive, symmetric)

- The relation $\iff$ on logical formulas.
- The relation 'have the same prime number divisors' on $\mathbb{N}$.
- The relation 'are citizens of the same country' on people.

Note

- An antisymmetric pre-order is a partial order; a symmetric pre-order is an equivalence relation.
- For convenience, also the pair (M, R) is called a pre-order, partial order, and strict partial order, respectively.
- More accurately, we could speak of the pair (M, R) as of a set M which is pre-ordered, partially ordered, and strictly partially ordered by R, respectively.
- Synonymously, we also speak of M as a pre-ordered, partially ordered, and a strictly partially ordered set, respectively, or of M as a set which is equipped with a pre-order, partial order and strict partial order, respectively.
- On any set, the equality relation $=$ is a partial order, called the discrete (partial) order.

The Strict Part of an Ordering

Let $\sqsubseteq$ be a pre-order (reflexive, transitive) on P.

Definition A.2.1.4 (Strict Part of $\sqsubseteq$)

The relation $\sqsubset$ on P defined by

$$\forall\, p, q \in P.\ p \sqsubset q \iff_{df} p \sqsubseteq q \ \wedge\ p \neq q$$

is called the strict part of $\sqsubseteq$.

Corollary A.2.1.5 (Strict Partial Order)

Let $(P, \sqsubseteq)$ be a partial order, let $\sqsubset$ be the strict part of $\sqsubseteq$.

Then: $(P, \sqsubset)$ is a strict partial order.

Useful Results

Let $\sqsubset$ be a strict partial order (asymmetric, transitive) on P.

Lemma A.2.1.6

The relation $\sqsubset$ is irreflexive.

Lemma A.2.1.7

The pair $(P, \sqsubseteq)$, where $\sqsubseteq$ is defined by

$$\forall\, p, q \in P.\ p \sqsubseteq q \iff_{df} p \sqsubset q \ \vee\ p = q$$

is a partial order.

Induced (or Inherited) Partial Order

Definition A.2.1.8 (Induced Partial Order)

Let $(P, \sqsubseteq_P)$ be a partially ordered set, let $Q \subseteq P$ be a subset of P, and let $\sqsubseteq_Q$ be the relation on Q defined by

$$\forall\, q, r \in Q.\ q \sqsubseteq_Q r \iff_{df} q \sqsubseteq_P r$$

Then: $\sqsubseteq_Q$ is called the induced partial order on Q (or the inherited order from P on Q).

Exercise A.2.1.9

Let $|$ denote the divisibility relation on the set of natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \,|\, 35$.

Prove or disprove: The divisibility relation $|$ on $\mathbb{N}_0$ is a

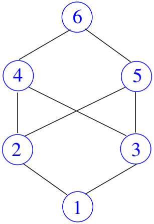
1. pre-order
2. partial order
3. strict partial order
4. equivalence (relation)

Proof or counterexample.

A.2.2
Hasse Diagrams

Hasse Diagrams

...are a sparse graphical representation of partial orders.



The links of a Hasse diagram

- are read from below to above (lower means smaller).
- represent the relation R of '$\cdot$ is an immediate predecessor of $\cdot$' defined by
  $p\,R\,q \iff_{df} \; p \sqsubset q \wedge \nexists\, r \in P.\; p \sqsubset r \sqsubset q$
  of a partial order $(P, \sqsubseteq)$, where $\sqsubset$ is the strict part of $\sqsubseteq$.

Reading Hasse Diagrams

The Hasse diagram representation of a partial order

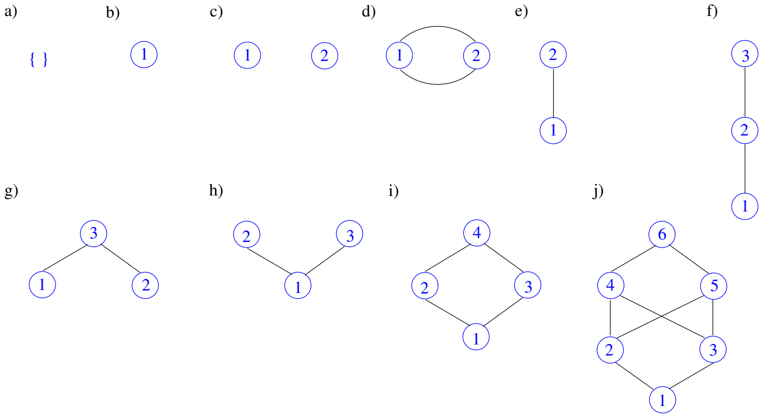
- omits links which express reflexive and transitive relations explicitly
- focuses on the 'immediate predecessor' relation.

The representation of a partial order by its Hasse diagram

- is sparse and thus economical (in the number of links).
- while preserving all relevant information of the partial order it represents:
  - $p \sqsubseteq q \land p = q$ (reflexivity): trivially represented (just without an explicit link)
  - $p \sqsubseteq q \land p \neq q$ (transitivity): represented by ascending paths (with at least one link) from p to q.

Exercise A.2.2.1

Which of the below diagrams are Hasse diagrams representing a partial order?



Exercise A.2.2.2

Let $|$ denote the divisibility relation on the set of natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \mid 35$.

Draw an expressive section of the Hasse diagram of the divisibility relation $|$ on $\mathbb{N}_0$.

A.2.3

Bounds and Extremal Elements

Bounds in Pre-Orders

Definition A.2.3.1 (Bounds in Pre-Orders)

Let $(Q, \sqsubseteq)$ be a pre-order, let $q \in Q$ and $Q' \subseteq Q$.

q is called a

- lower bound of Q', in signs: $q \sqsubseteq Q'$, if $\forall\, q' \in Q'.\ q \sqsubseteq q'$
- upper bound of Q', in signs: $Q' \sqsubseteq q$, if $\forall\, q' \in Q'.\ q' \sqsubseteq q$
- greatest lower bound (glb) (or infimum) of Q', in signs: $\bigsqcap Q'$, if q is a lower bound of Q' and for every other lower bound $\hat{q}$ of Q' holds: $\hat{q} \sqsubseteq q$.
- least upper bound (lub) (or supremum) of Q', in signs: $\bigsqcup Q'$, if q is an upper bound of Q' and for every other upper bound $\hat{q}$ of Q' holds: $q \sqsubseteq \hat{q}$.

Extremal Elements in Pre-Orders

Definition A.2.3.2 (Extremal Elements in Pre-Ord's)

Let $(Q, \sqsubseteq)$ be a pre-order, let $\sqsubset$ be the strict part of $\sqsubseteq$, and let $Q' \subseteq Q$ and $q \in Q'$.

q is called a

- minimal element of Q', if there is no $q' \in Q'$ with $q' \sqsubset q$.
- maximal element of Q', if there is no $q' \in Q'$ with $q \sqsubset q'$.
- least (or minimum) element of Q', if $q \sqsubseteq Q'$.
- greatest (or maximum) element of Q', if $Q' \sqsubseteq q$.

Note: Least and greatest elements of Q itself are usually denoted by $\bot$ and $\top$ (bottom, top (in German: Tief, Hoch)), respectively, if they exist. Least (greatest) elements of Q are always minimal (maximal) elements of Q.

Existence and Uniqueness

...of bounds and extremal elements in partially ordered sets.

Let $(P, \sqsubseteq)$ be a partial order, and let $Q \subseteq P$ be a subset of P.

Lemma A.2.3.3 (lub/glb: Unique if Existent)

Least upper bounds, greatest lower bounds, least elements, and greatest elements in Q are unique, if they exist.

Lemma A.2.3.4 (Minimal/Maximal El.: Not Unique)

Minimal and maximal elements in Q are usually not unique.

Note: Lemma A.2.3.3 suggests considering $\bigsqcup$ and $\bigsqcap$ partial maps $\bigsqcup, \bigsqcap : \mathcal{P}(P) \to P$ from the powerset $\mathcal{P}(P)$ of P to P.
Lemma A.2.3.3 does not hold for pre-orders.

Characterization of Least, Greatest Elements

...in terms of infima and suprema of sets.

Let $(P, \sqsubseteq)$ be a partial order.

Lemma A.2.3.5 (Characterization of $\bot$ and $\top$)

The least element $\bot$ and the greatest element $\top$ of P are given by the supremum and the infimum of the empty set, and the infimum and the supremum of P, respectively, i.e.,

$$\bot = \bigsqcup \emptyset = \bigsqcap P \quad \text{and} \quad \top = \bigsqcap \emptyset = \bigsqcup P$$

if they exist.

Lower and Upper Bound Sets

Considering $\bigsqcup$ and $\bigsqcap$ partial functions $\bigsqcup, \bigsqcap : \mathcal{P}(P) \to P$ on the powerset of a partial order $(P, \sqsubseteq)$ suggests introducing two further maps $LB, UB : \mathcal{P}(P) \to \mathcal{P}(P)$ on $\mathcal{P}(P)$:

Definition A.2.3.6 (Lower and Upper Bound Sets)

Let $(P, \sqsubseteq)$ be a partial order. Then:

$LB, UB : \mathcal{P}(P) \to \mathcal{P}(P)$ denote two maps, which map a subset $Q \subseteq P$ to the set of its lower bounds and upper bounds, respectively:

1. $\forall\, Q \subseteq P.\; LB(Q) =_{df} \{lb \in P \mid\; lb \sqsubseteq Q\}$
2. $\forall\, Q \subseteq P.\; UB(Q) =_{df} \{ub \in P \mid Q \sqsubseteq ub\}$

Properties of Lower and Upper Bound Sets

Lemma A.2.3.7

Let $(P, \sqsubseteq)$ be a partial order, and let $Q \subseteq P$. Then:

$$\bigsqcup Q = \bigsqcap UB(Q) \;\text{ and }\; \bigsqcap Q = \bigsqcup LB(Q)$$

if the supremum and the infimum of Q exist.

Lemma A.2.3.8

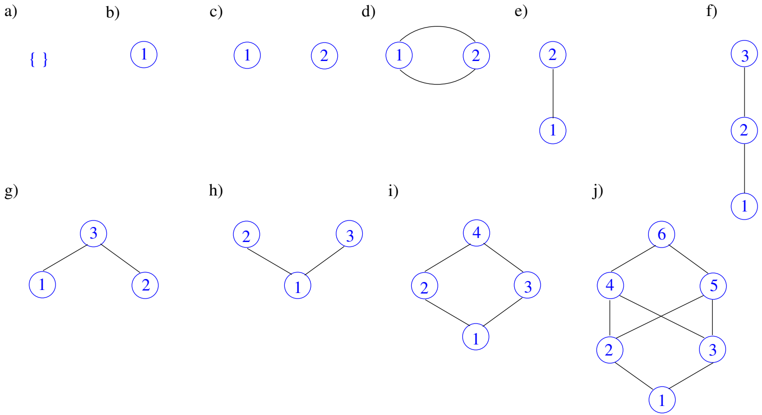
Let $(P, \sqsubseteq)$ be a partial order, and let $Q, Q_1, Q_2 \subseteq P$. Then:

1. $Q_1 \subseteq Q_2 \Rightarrow LB(Q_1) \supseteq LB(Q_2) \wedge UB(Q_1) \supseteq UB(Q_2)$
2. $UB(LB(UB(Q))) = UB(Q)$
3. $LB(UB(LB(Q))) = LB(Q)$

Note: Lemma A.2.3.8(1) shows that LB and UB are antitonic maps (cf. Chapter A.2.7).

Exercise A.2.3.9

Which of the elements of the below diagrams are minimal, maximal, least or greatest?



Exercise A.2.3.10

Let $|$ denote the divisibility relation on the set of natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \mid 35$.

Write down the sets of elements of $\mathbb{N}_0$, which are

1. minimal
2. maximal
3. least
4. greatest

wrt the divisibility relation $|$ on $\mathbb{N}_0$.

A.2.4

Noetherian and Artinian Orders

Noetherian and Artinian Orders

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.2.4.1 (Noetherian Order)

$(P, \sqsubseteq)$ is called a Noetherian order, if every non-empty subset $\emptyset \neq Q \subseteq P$ contains a minimal element.

Definition A.2.4.2 (Artinian Order)

$(P, \sqsubseteq)$ is called an Artinian order, if the dual order $(P, \sqsupseteq)$ of $(P, \sqsubseteq)$ is a Noetherian order.

Lemma A.2.4.3

$(P, \sqsubseteq)$ is an Artinian order iff every non-empty subset $\emptyset \neq Q \subseteq P$ contains a maximal element.

Well-founded Orders

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.2.4.4 (Well-founded Order)

$(P, \sqsubseteq)$ is called a well-founded order, if $(P, \sqsubseteq)$ is a Noetherian order and totally ordered.

Lemma A.2.4.5

$(P, \sqsubseteq)$ is a well-founded order iff every non-empty subset $\emptyset \neq Q \subseteq P$ contains a least element.

Noetherian Induction

Theorem A.2.4.6 (Noetherian Induction)

Let $(N, \sqsubseteq)$ be a Noetherian order, let $N_{min} \subseteq N$ be the set of minimal elements of N, and let $\phi : N \to \mathbb{B}$ be a predicate on N. Then:

If

1. $\forall\, n \in N_{min}.\ \phi(n)$ (Induction base)
2. $\forall\, n \in N \backslash N_{min}.\ (\forall\, m \sqsubset n.\ \phi(m)) \Rightarrow \phi(n)$ (Induction step)

then:

$$\forall\, n \in N.\ \phi(n)$$

A.2.5

Chains

Chains, Antichains

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.2.5.1 (Chain)

A set $C \subseteq P$ is called a chain, if the elements of C are totally ordered, i.e., $\forall\, c_1, c_2 \in C.\; c_1 \sqsubseteq c_2 \vee c_2 \sqsubseteq c_1$.

Definition A.2.5.2 (Antichain)

A set $C \subseteq P$ is called an antichain, if $\forall\, c_1, c_2 \in C.\; c_1 \sqsubseteq c_2 \Rightarrow c_1 = c_2$.

Definition A.2.5.3 (Finite, Infinite (Anti-) Chain)

Let $C \subseteq P$ be a chain or an antichain. C is called finite, if the number of its elements is finite; C is called infinite otherwise.

Note: Any set P may be converted into an antichain by giving it the discrete order: $(P, =)$.

Ascending Chains, Descending Chains

Definition A.2.5.4 (Ascending, Descending Chain)

Let $C \subseteq P$ be a chain. C given in the form of

- $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}$
- $C = \{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\}$

is called an ascending chain and descending chain, respectively.

Examples of Chains

The set

- $S =_{df} \{n \in \mathbb{N} \mid n \ even\}$ is a chain in $\mathbb{N}$.
- $S =_{df} \{z \in \mathbb{Z} \mid z \ odd\}$ is a chain in $\mathbb{Z}$.
- $S =_{df} \{\, \{k \in \mathbb{N} \mid k < n\} \mid n \in \mathbb{N}\}$ is a chain in the powerset $\mathcal{P}(\mathbb{N})$ of $\mathbb{N}$.

Note: A chain can always be given in the form of an ascending or descending chain.

- $\{0 \leq 2 \leq 4 \leq 6 \leq \ldots\}$: $\mathbb{N}$ as ascending chain.
- $\{\ldots \geq 6 \geq 4 \geq 2 \geq 0\}$: $\mathbb{N}$ as descending chain.
- $\{\ldots \leq -3 \leq -1 \leq 1 \leq 3 \leq \ldots\}$: $\mathbb{Z}$ as ascending chain.
- $\{\ldots \geq 3 \geq 1 \geq -1 \geq -3 \geq \ldots\}$: $\mathbb{Z}$ as descending chain.
- ...

Eventually Stationary Sequences

Definition A.2.5.5 (Stationary Sequence)

1. An ascending sequence of the form
$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots$$
is called eventually stationary, if
$$\exists\, n \in \mathbb{N}.\ \forall\, j \in \mathbb{N}.\ p_{n+j} = p_n$$
2. A descending sequence of the form
$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots$$
is called eventually stationary, if
$$\exists\, n \in \mathbb{N}.\ \forall\, j \in \mathbb{N}.\ p_{n+j} = p_n$$

Chains and Sequences

Lemma A.2.5.6

An ascending or descending sequence of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots$$

1. is a finite chain iff it is eventually stationary.
2. is an infinite chain iff it is not eventually stationary.

Note the subtle difference between the notion of chains in terms of sets

$$\{p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots\} \;\; \text{or} \quad \{p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots\}$$

and in terms of sequences

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots$$

Sequences may contain duplicates, which would correspond to defining chains in terms of multisets.

Ascending, Descending Chain Condition

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.2.5.7 (Asc./Desc. Chain Condition)

$(P, \sqsubseteq)$ satisfies the

1. ascending chain condition (in German: aufsteigende Kettenbedingung), if every ascending chain is eventually stationary, i.e., for every chain $p_1 \sqsubseteq p_2 \sqsubseteq \ldots \sqsubseteq p_n \sqsubseteq \ldots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.
2. descending chain condition (in German: absteigende Kettenbedingung), if every descending chain is eventually stationary, i.e., for every chain $p_1 \sqsupseteq p_2 \sqsupseteq \ldots \sqsupseteq p_n \sqsupseteq \ldots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.

Chains and Noetherian Orders

Let $(P, \sqsubseteq)$ be a partial order.

Lemma A.2.5.8 (Noetherian Order)

The following statements are equivalent:

1. $(P, \sqsubseteq)$ is a Noetherian order.
2. $(P, \sqsubseteq)$ satisfies the descending chain condition.
3. Every chain of the form
$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots$$
is eventually stationary, i.e.: $\exists\, n \in \mathbb{N}.\ \forall\, j \in \mathbb{N}.\ p_{n+j} = p_n$.
4. Every chain of the form
$$p_0 \sqsupset p_1 \sqsupset p_2 \sqsupset \ldots$$
is finite.

Chains and Artinian Orders

Let $(P, \sqsubseteq)$ be a partial order.

Lemma A.2.5.9 (Artinian Order)

The following statements are equivalent:

1. $(P, \sqsubseteq)$ is an Artinian order.
2. $(P, \sqsubseteq)$ satisfies the ascending chain condition.
3. Every chain of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots$$

   is eventually stationary, i.e.: $\exists\, n \in \mathbb{N}.\ \forall\, j \in \mathbb{N}.\ p_{n+j} = p_n$.
4. Every chain of the form

$$p_0 \sqsubset p_1 \sqsubset p_2 \sqsubset \ldots$$

   is finite.

Chains and Noetherian, Artinian Orders

Let $(P, \sqsubseteq)$ be a partial order.

Lemma A.2.5.10 (Noetherian and Artinian Order)

The following statements are equivalent:

1. $(P, \sqsubseteq)$ is a Noetherian and an Artinian order.
2. $(P, \sqsubseteq)$ satisfies the descending and the ascending chain condition.
3. Every chain $C \subseteq P$ is finite.

A.2.6

Directed Sets

Directed Sets

Let $(P, \sqsubseteq)$ be a partial order, and let $\emptyset \neq D \subseteq P$.

Definition A.2.6.1 (Directed Set)

D $(\neq \emptyset)$ is called a directed set (in German: gerichtete Menge), if

$$\forall\, d, e \in D.\ \exists\, f \in D.\ f \in UB(\{d, e\})$$

i.e., for any two elements d and e there is a common upper bound of d and e in D, i.e., $UB(\{d, e\}) \cap D \neq \emptyset$.

Properties of Directed Sets

Let $(P, \sqsubseteq)$ be a partial order, and let $D \subseteq P$.

Lemma A.2.6.2

D is a directed set iff any finite subset $D' \subseteq D$ has an upper bound in D, i.e., $\exists\, d \in D.\ d \in UB(D')$, i.e., $UB(D') \cap D \neq \emptyset$.

Lemma A.2.6.3

If D has a greatest element, then D is a directed set.

Properties of Finite Directed Sets

Let $(P, \sqsubseteq)$ be a partial order, and let $D \subseteq P$.

Corollary A.2.6.4

Let D be a finite directed set. Then: $\bigsqcup D$ *exists* $\in D$ and is the greatest element of D.

Proof. Since D a directed set, we have:

$$\exists\, d \in D.\ d \in UB(D), \text{ i.e., } UB(D) \cap D \neq \emptyset.$$

This means $D \sqsubseteq d$. The antisymmetry of $\sqsubseteq$ yields that the element enjoying this property is unique. Thus, d is the (unique) greatest element of D given by $\bigsqcup D$, i.e., $d = \bigsqcup D$.

Note: If D is infinite, the statement of Corollary A.2.6.4 does usually not hold.

Strongly Directed Sets

Let $(P, \sqsubseteq)$ be a partial order with least element $\perp$, and let $D \subseteq P$.

Definition A.2.6.5 (Strongly Directed Set)

$D \neq \emptyset$ is called a strongly directed set (in German: stark gerichtete Menge), if

1. $\perp \in D$
2. $\forall\, d, e \in D.\ \exists\, f \in D.\ f = \bigsqcup\{d, e\}$, i.e., for any two elements d and e the supremum $\bigsqcup\{d, e\}$ of d and e exists in D.

Properties of Strongly Directed Sets

Let $(P, \sqsubseteq)$ be a partial order with least element $\perp$, and let $D \subseteq P$.

Lemma A.2.6.6

D is a strongly directed set iff every finite subset $D' \subseteq D$ has a supremum in D, i.e., $\exists\, d \in D.\ d = \bigsqcup D'$.

Lemma A.2.6.7

Let D be a finite strongly directed set. Then: $\bigsqcup D$ *exists* $\in D$ and is the greatest element of D.

Note: The statement of Lemma A.2.6.7 does usually not hold, if D is infinite.

Directed Sets, Strongly Directed Sets, Chains

Let $(P, \sqsubseteq)$ be a partial order with least element $\perp$.

Lemma A.2.6.8

Let $\emptyset \neq D \subseteq P$ be a non-empty subset of P. Then:

1. D is a directed set, if D is a strongly directed set.
2. D is a strongly directed set, if $\perp \in D$ and D is a chain.

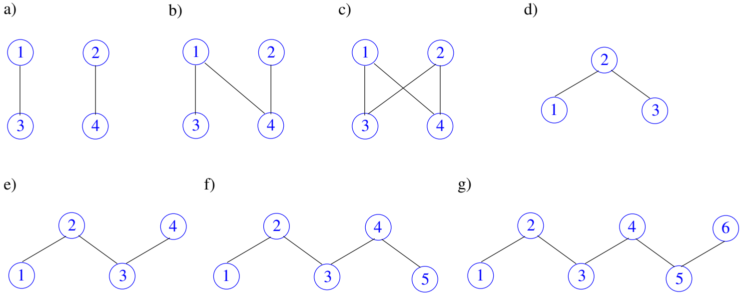
Corollary A.2.6.9

Let $\emptyset \neq D \subseteq P$ be a non-empty subset of P. Then:

$\perp \in D \wedge D$ chain $\Rightarrow D$ strongly directed set $\Rightarrow D$ directed set

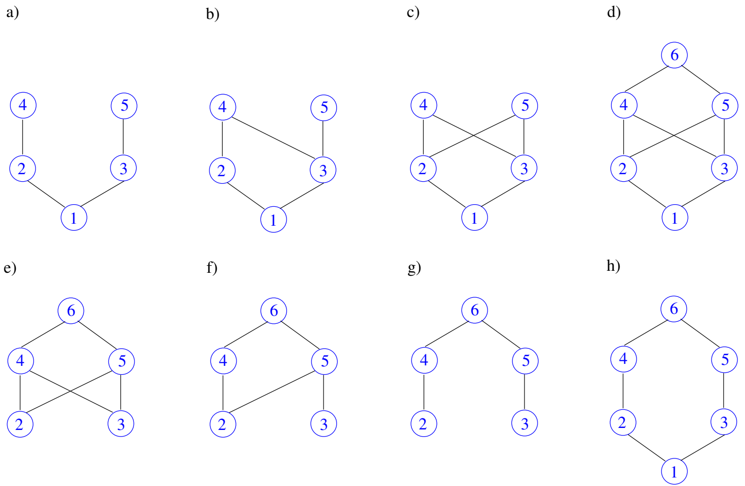
Exercise A.2.6.10

Which of the below partial orders are (strongly) directed sets?
Which of their subsets are (strongly) directed sets?



Exercise A.2.6.11

Which of the below partial orders are (strongly) directed sets?
Which of their subsets are (strongly) directed sets?



Exercise A.2.6.12

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq \;=_{df}\; |$, where $|$ denotes the divisibility relation on the natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \,|\, 35$.

Is the set $\mathbb{N}_0$

1. directed?
2. strongly directed?

What subsets of $\mathbb{N}_0$ are

1. directed?
2. strongly directed?

Proof or counterexample.

A.2.7

Maps on Partial Orders

Monotonic and Antitonic Maps on POs

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be partial orders, and let $f \in [C \to D]$ be a map from C to D.

Definition A.2.7.1 (Monotonic Maps on POs)

f is called monotonic (or order preserving) iff

$$\forall\, c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$$

(Preservation of the ordering of elements)

Definition A.2.7.2 (Antitonic Maps on POs)

f is called antitonic (or order inversing) iff

$$\forall\, c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c') \sqsupseteq_D f(c)$$

(Inversion of the ordering of elements)

Expanding and Contracting Maps on POs

Let $(C, \sqsubseteq_C)$ be a partial order, let $f \in [C \to C]$ be a map on C, and let $\hat{c} \in C$ be an element of C.

Definition A.2.7.3 (Expanding Maps on POs)

f is called

- expanding (or inflationary) for $\hat{c}$ iff $\hat{c} \sqsubseteq f(\hat{c})$
- expanding (or inflationary) iff $\forall\, c \in C.\ c \sqsubseteq f(c)$

Definition A.2.7.4 (Contracting Maps on POs)

f is called

- contracting (or deflationary) for $\hat{c}$ iff $f(\hat{c}) \sqsubseteq \hat{c}$
- contracting (or deflationary) iff $\forall\, c \in C.\ f(c) \sqsubseteq c$

A.2.8

Order Homomorphisms, Order Isormorphisms

PO Homomorphisms, PO Isomorphisms

Let $(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ be partial orders, and let $f \in [P \to R]$ be a map from P to R.

Definition A.2.8.1 (PO Hom. & Isomorphism)

f is called an

1. order homomorphism between P and R, if f is monotonic (or order preserving), i.e.,
$$\forall\, p, q \in P.\ p \sqsubseteq_P q \Rightarrow f(p) \sqsubseteq_R f(q)$$
2. order isomorphism between P and R, if f is a bijective order homomorphism between P and R and the inverse f^{-1} of f is an order homomorphism between R and P.

Definition A.2.8.2 (Order Isomorphic)

$(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ are called order isomorphic, if there is an order isomorphism between P and R.

PO Embeddings

Let $(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ be partial orders, and let $f \in [P \to R]$ be a map from P to R.

Definition A.2.8.3 (PO Embedding)

f is called an order embedding of P in R iff

$$\forall\, p, q \in P.\ p \sqsubseteq_P q \iff f(p) \sqsubseteq_R f(q)$$

Lemma A.2.8.4 (PO Embeddings and Isomorphisms)

f is an order isomorphism between P and R iff f is an order embedding of P in R and f is surjective.

Intuitively: Partial orders, which are order isomorphic, are 'essentially the same.'

A.3

Complete Partially Ordered Sets

A.3.1

Chain and Directly Complete Partial Orders

Complete Partially Ordered Sets

...or Complete Partial Orders:

- a slightly weaker ordering notion than that of a lattice (cf. Appendix A.4), which is often more adequate for the modelling of problems in computer science, where full lattice properties are often not required.
- come in two different flavours as so-called
  - Chain Complete Partial Orders (CCPOs)
  - Directedly Complete Partial Orders (DCPOs)

  based on the notions of chains and directed sets, respectively, which, however, are equivalent (cf. Theorem 3.1.7).

Complete Partial Orders: CCPO View

Definition A.3.1.1 (Chain Complete Partial Order)

A partial order $(P, \sqsubseteq)$ is a

1. chain complete partial order (pre-CCPO), if every non-empty (ascending) chain $\emptyset \neq C \subseteq P$ has a least upper bound $\bigsqcup C$ in P, i.e., $\bigsqcup C$ *exists* $\in P$.
2. pointed chain complete partial order (CCPO), if every (ascending) chain $C \subseteq P$ has a least upper bound $\bigsqcup C$ in P, i.e., $\bigsqcup C$ *exists* $\in P$.

Note: Some authors use CCPO and CCPPO instead of pre-CCPO and CCPO, respectively.

Complete Partial Orders: DCPO View

Definition A.3.1.2 (Directedly Complete Partial Ord.)

A partial order $(P, \sqsubseteq)$ is a

1. directedly complete partial order (pre-DCPO), if every directed subset $D \subseteq P$ has a least upper bound $\bigsqcup D$ in P, i.e., $\bigsqcup D$ *exists* $\in P$.
2. pointed directedly complete partial order (DCPO), if it is a pre-DCPO and has a least element $\perp$.

Note: Some authors use DCPO and DCPPO instead of pre-DCPO and DCPO, respectively.

Remarks on CCPOs and DCPOs

On CCPOs:

- A CCPO is often called a domain.
- 'Ascending chain' and 'chain' can equivalently be used in Definition A.3.1.1, since a chain can always be given in ascending order. 'Ascending' chain is just more intuitive.

On DCPOs:

- A directed set S, in which by definition every finite subset has an upper bound in S, does not need to have a supremum in S, if S is infinite. Therefore, the DCPO property does not trivially follow from the directed set property (cf. Corollary A.2.6.4).

Existence of Least Elements in CPOs

Lemma A.3.1.3 (Least Elem. Existence in CPOs)

Let $(C, \sqsubseteq)$ be a CPO, i.e., a CCPO or DCPO. Then there is a unique least element in C, denoted by $\bot$, which is given by the supremum of the empty chain or set, i.e.: $\bot = \bigsqcup \emptyset$.

Corollary A.3.1.4 (Non-Emptyness of CPOs)

Let $(C, \sqsubseteq)$ be a CPO, i.e., a CCPO or DCPO. Then: $C \neq \emptyset$.

Note: Lemma A.3.1.3 does not hold for pre-CPOs, i.e., a pre-CPO $(P, \sqsubseteq)$ does not need to have a least element.

Relating Finite POs, CCPOs and DCPOs

Let P be a finite set, and let $\sqsubseteq$ be a relation on P.

Lemma A.3.1.5 (Fin. POs, pre-CCPOs, pre-DCPOs)

The following statements are equivalent:

1. $(P, \sqsubseteq)$ is a partial order.
2. $(P, \sqsubseteq)$ is a pre-CCPO.
3. $(P, \sqsubseteq)$ is a pre-DCPO.

Lemma A.3.1.6 (Finite POs, CCPOs, DCPOs)

Let $p \in P$ with $p \sqsubseteq P$. Then the following statements are equivalent:

1. $(P, \sqsubseteq)$ is a partial order.
2. $(P, \sqsubseteq)$ is a CCPO.
3. $(P, \sqsubseteq)$ is a DCPO.

Equivalence of CCPOs and DCPOs

Theorem A.3.1.7 (Equivalence)

Let $(P, \sqsubseteq)$ be a partial order. Then the following statements are equivalent:

1. $(P, \sqsubseteq)$ is a CCPO.
2. $(P, \sqsubseteq)$ is a DCPO.

Note: We simply speak of a CPO, if its flavour based on chains (CCPO) or directed sets (DCPO) does not matter; analogously, this applies to pre-CPOs.

Examples of pre-CPOs and CPOs (1)

- $(\mathcal{P}(\mathbb{N}), \subseteq)$ is a CPO (i.e., a CCPO and a DCPO).
  - Least element: $\emptyset$
  - Least upper bound $\bigsqcup C$ of C *chain* $\subseteq \mathcal{P}(\mathbb{N})$: $\bigcup_{C' \in C} C'$
- The set of finite and infinite strings S partially ordered by the prefix relation $\sqsubseteq_{pfx}$ defined by

$$\forall\, s, s'' \in S.\ s \sqsubseteq_{pfx} s'' \iff_{df}$$
$$s = s'' \vee (s \ \textit{finite} \ \wedge \exists\, s' \in S.\ s +\!\!+ s' = s'')$$

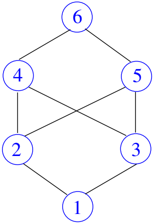
  is a CPO.
- $(\{-n \mid n \in \mathbb{N}\}, \leq)$ is a pre-CPO (i.e., a pre-CCPO and a pre-DCPO) but not a CPO (i.e., not a CCPO and DCPO).

Examples of pre-CPOs and CPOs (2)

- $(\emptyset, \emptyset)$ is a pre-CPO (i.e., a pre-CCPO and a pre-DCPO) but not a CPO (i.e., not a CCPO and DCPO).

  (Both the pre-CCPO (absence of non-empty chains in $\emptyset$) and the pre-DCPO ($\emptyset$ is the only subset of $\emptyset$ and is not directed by definition) property holds trivially. Note also that $P = \emptyset$ implies $\sqsubseteq\, = \emptyset \subseteq P \times P$).

- The partial order $(P\ \sqsubseteq)$ given by the below Hasse diagram is a CPO.



Examples of pre-CPOs and CPOs (3)

- The set of finite and infinite strings S partially ordered by the lexicographical order $\sqsubseteq_{lex}$ defined by

  $$\forall\, s,t \in S.\; s \sqsubseteq_{lex} t \iff_{df}$$
  $$s = t \vee (\exists\, p \; finite, s', t' \in S.\; s = p +\!\!+ s' \;\wedge\; t = p +\!\!+ t' \;\wedge\; (s' = \varepsilon \;\vee\; s'_1 < t'_1)\,)$$

  where ε denotes the empty string, $w\!\downarrow_1$ denotes the first character of a string w, and $<$ the lexicographical ordering on characters, is a CPO (i.e., a CCPO and a DCPO).

(Anti-) Examples of CPOs

- $(\mathbb{N}, \leq)$ is not a CPO (i.e., not a CCPO and DCPO).
- The set of finite strings S_{fin} partially ordered by the
  - prefix relation $\sqsubseteq_{pfx}$ defined by

    $$\forall\, s, s' \in S_{fin}.\ s \sqsubseteq_{pfx} s' \iff_{df} \exists\, s'' \in S_{fin}.\ s +\!\!+ s'' = s'$$

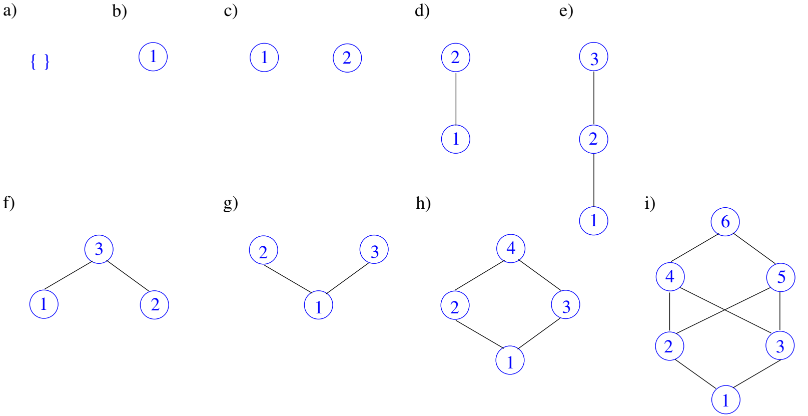
    is not a CPO (i.e., not a CCPO and DCPO).
  - lexicographical order $\sqsubseteq_{lex}$ defined by

    $$\forall\, s, t \in S_{fin}.\ s \sqsubseteq_{lex} t \iff_{df}$$
    $$\exists\, p, s', t' \in S_{fin}.\ s = p +\!\!+ s' \ \wedge\ t = p +\!\!+ t' \ \wedge$$
    $$(s' = \varepsilon \ \vee\ s'\!\downarrow_1 \ < t'\!\downarrow_1)$$

    where ε denotes the empty string, $w\!\downarrow_1$ denotes the first character of a string w, and $<$ the lexicographical ordering on characters, is not a CPO (i.e., not a CCPO and DCPO).
- $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a CPO (i.e., not a CCPO and DCPO).

Exercise A.3.1.8

Which of the partial orders given by the below Hasse diagrams are (pre-) CCPOs? Which ones are (pre-) DCPOs?



Strongly Directed CPOs: A DCPO Variant

On DCPOs based on Strongly Directed Sets

- Replacing directed sets by strongly directed sets in Definition A.3.1.2 leads to SDCPOs.
- Recalling that strongly directed sets are not empty (cf. Definition A.2.6.5), there is no analogue of pre-DCPOs for strongly directed sets.
- A strongly directed set S, in which by definition every finite subset has a supremum in S, does not need to have a supremum itself in S, if S is infinite. Therefore, the SDCPO property does not trivially follow from the strongly directed property of sets (cf. Corollary A.2.6.4).

Exercise A.3.1.9

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \,|\, 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. pre-CCPO
2. CCPO
3. pre-DCPO
4. DCPO
5. SDCPO

Proof or counterexample.

A.3.2

Maps on Complete Partial Orders

Continuous Maps on CCPOs

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be CCPOs, and let $f \in [C \to D]$ be a map from C to D.

Definition A.3.2.1 (Continuous Maps on CCPOs)

f is called continuous iff f is monotonic and

$$\forall\, C' \neq \emptyset \; chain \subseteq C.\; f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$$

(Preservation of least upper bounds)

Note: $\forall\, S \subseteq C.\; f(S) =_{df} \{\, f(s) \mid s \in S \}$

Continuous Maps on DCPOs

Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be DCPOs, and let $f \in [D \to E]$ be a map from D to E.

Definition A.3.2.2 (Continuous Maps on DCPOs)

f is called continuous iff

$$\forall\, D' \neq \emptyset \textit{ directed set} \subseteq D.\ f(D') \text{ directed set } \subseteq E \,\wedge$$
$$f(\bigsqcup_D D') =_E \bigsqcup_E f(D')$$

(Preservation of least upper bounds)

Note: $\forall\, S \subseteq D.\ f(S) =_{df} \{\, f(s) \mid s \in S \}$

Characterizing Monotonicity

Let $(C, \sqsubseteq_C), (D, \sqsubseteq_D)$ be CCPOs, let $(E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be DCPOs.

Lemma A.3.2.3 (Characterizing Monotonicity)

1. $f : C \to D$ is monotonic
   iff $\forall\, C' \neq \emptyset\ \mathit{chain} \subseteq C.$
   $$f(C')\ \mathit{chain} \subseteq D \wedge f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$$
2. $g : E \to F$ is monotonic
   iff $\forall\, E' \neq \emptyset\ \mathit{directed\ set} \subseteq E.$
   $$g(E')\ \mathit{directed\ set} \subseteq F \wedge g(\bigsqcup_E E') \sqsupseteq_F \bigsqcup_F g(E')$$

Strict Maps on CCPOs and DCPOs

Let $(C, \sqsubseteq_C), (D, \sqsubseteq_D)$ be CCPOs with least elements $\perp_C$ and $\perp_D$, respectively, let $(E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be DCPOs with least elements $\perp_E$ and $\perp_F$, respectively, and let $f \in [C \overset{con}{\to} D]$ and $g \in [E \overset{con}{\to} F]$ be continuous maps.

Definition A.3.2.4 (Strict Functions on CPOs)

f and g are called strict, if the equalities

- $f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$, $g(\bigsqcup_E E') =_F \bigsqcup_F g(E')$

also hold for $C' = \emptyset$ and $E' = \emptyset$, i.e., if the equalities

- $f(\bigsqcup_C \emptyset) =_C \;\; f(\perp_C) =_D \perp_D \;\; =_D \bigsqcup \emptyset$
- $f(\bigsqcup_E \emptyset) =_E \;\; g(\perp_E) =_F \perp_F \;\; =_F \bigsqcup \emptyset$

are valid.

A.3.3

Mechanisms for Constructing Complete Partial Orders

Common CCPO and DCPO Constructions

The following construction principles hold for

- CCPOs
- DCPOs

Therefore, we simply write CPO.

Common CPO Constructions: Flat CPOs

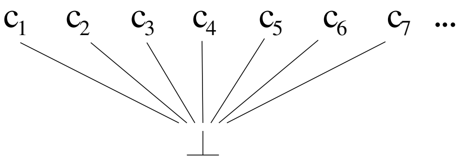
Lemma A.3.3.1 (Flat CPO Construction)

Let C be a set. Then:

$(C \,\dot\cup\, \{\bot\}, \sqsubseteq_{flat})$ with $\sqsubseteq_{flat}$ defined by

$$\forall\, c, d \in C \,\dot\cup\, \{\bot\}.\; c \sqsubseteq_{flat} d \iff_{df} c = \bot \;\vee\; c = d$$

is a CPO, a so-called flat CPO.



Common CPO Constructions: Flat pre-CPOs

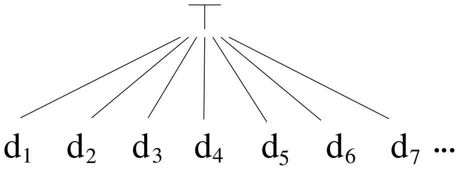
Lemma A.3.3.2 (Flat Pre-CPO Construction)

Let D be a set. Then:

$(D \dot\cup \{\top\}, \sqsubseteq_{flat})$ with $\sqsubseteq_{flat}$ defined by

$$\forall\, d, e \in D \dot\cup \{\top\}.\ d \sqsubseteq_{flat} e \iff_{df} e = \top \ \vee\ d = e$$

is a pre-CPO, a so-called flat pre-CPO.



Common CPO Constructions: Products (1)

Lemma A.3.3.3 (Non-strict Product Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The non-strict product $(\times P_i, \sqsubseteq_\times)$, where

- $\times P_i =_{df} P_1 \times P_2 \times \ldots \times P_n$ is the cartesian product of all P_i, $1 \leq i \leq n$
- $\sqsubseteq_\times$ is defined pointwise by

$$\forall\, (p_1, \ldots, p_n), (q_1, \ldots, q_n) \in \times P_i.$$
$$(p_1, \ldots, p_n) \sqsubseteq_\times (q_1, \ldots, q_n) \iff_{df} \forall\, i \in \{1, \ldots, n\}.\; p_i \sqsubseteq_i q_i$$

is a CPO.

Common CPO Constructions: Products (2)

Lemma A.3.3.4 (Strict Product Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The strict (or smash) product $(\bigotimes P_i, \sqsubseteq_\otimes)$, where

- $\bigotimes P_i =_{df} \times P_i$ is the the cartesian product of all P_i
- $\sqsubseteq_\otimes =_{df} \sqsubseteq_\times$ defined pointwise with the additional setting

$$(p_1, \ldots, p_n) = \bot \iff_{df} \exists\, i \in \{1, \ldots, n\}.\ p_i = \bot_i$$

is a CPO.

Common CPO Constructions: Sums (1)

Lemma A.3.3.5 (Separated Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The separated (or direct) sum $(\bigoplus_\perp P_i, \sqsubseteq_{\oplus_\perp})$, where

- $\bigoplus_\perp P_i =_{df} P_1 \dot\cup P_2 \,\dot\cup \ldots \dot\cup\, P_n \,\dot\cup\, \{\perp\}$ is the disjoint union of all P_i, $1 \leq i \leq n$, and a fresh bottom element $\perp$
- $\sqsubseteq_{\oplus_\perp}$ is defined by

  $\forall\, p, q \in \bigoplus_\perp P_i.\ p \sqsubseteq_{\oplus_\perp} q \iff_{df}$

  $$p = \perp \ \vee\ (\exists\, i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q)$$

is a CPO.

Common CPO Constructions: Sums (2)

Lemma A.3.3.6 (Coalesced Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The coalesced sum $(\bigoplus_\vee P_i, \sqsubseteq_{\oplus\vee})$, where

- $\bigoplus_\vee P_i =_{df} P_1 \backslash \{\perp_1\} \dot\cup P_2 \backslash \{\perp_2\} \,\dot\cup \ldots \dot\cup\, P_n \backslash \{\perp_n\} \,\dot\cup\, \{\perp\}$ is the disjoint union of all P_i, $1 \leq i \leq n$, and a fresh bottom element $\perp$, which is identified with and replaces the least elements $\perp_i$ of the sets P_i, i.e., $\perp =_{df} \perp_i$, $i \in \{1, \ldots, n\}$
- $\sqsubseteq_{\oplus\vee}$ is defined by
$$\forall\, p, q \in \bigoplus_\vee P_i.\ p \sqsubseteq_{\oplus\vee} q \iff_{df}$$
$$p = \perp \ \vee\ (\exists\, i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q)$$

is a CPO.

Common CPO Constructions: Function Space

Lemma A.3.3.7 (Continuous Function Space Con.)

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be pre-CPOs. Then:

The continuous function space $([C \stackrel{con}{\to} D], \sqsubseteq_{cfs})$, where

- $[C \stackrel{con}{\to} D]$ is the set of continuous maps from C to D
- $\sqsubseteq_{cfs}$ is defined pointwise by
  $$\forall\, f, g \in [C \stackrel{con}{\to} D].\ f \sqsubseteq_{cfs} g \iff_{df} \forall\, c \in C.\ f(c) \sqsubseteq_D g(c)$$

is a pre-CPO. It is a CPO, if $(D, \sqsubseteq_D)$ is a CPO.

Note: The definition of $\sqsubseteq_{cfs}$ does not make use of C being a pre-CPO. This requirement is only to allow us tailoring the definition to continuous maps.

Applications of CPOs

...in functional programming:

- Flat CPOs: Modelling, ordering the values of, e.g., the polymorphic type `Maybe a`.
- Non-strict Product CPOs: Modelling, ordering the values of tuple types, approximating the values of streams, modelling non-strict functions.
- Strict Product CPOs: Modelling, ordering the values of tuple types, modeling strict functions.
- Sum CPOs: Modelling, ordering the values of union types (called sum types in Haskell).
- Function-space CPOs: Defining the (denotational) semantics of programs.

A.4

Lattices

A.4.1

Lattices, Complete Lattices

Lattices and Complete Lattices

Let $(P, \sqsubseteq)$ be a partial order, $P \neq \emptyset$.

Definition A.4.1.1 (Lattice)

$(P, \sqsubseteq)$ is a lattice (in German: Verband), if every non-empty finite subset P' of P has a least upper bound and a greatest lower bound in P.

Definition A.4.1.2 (Complete Lattice)

$(P, \sqsubseteq)$ is a complete lattice (in German: vollständiger Verband), if every subset P' of P has a least upper bound and a greatest lower bound in P.

Note: Lattices and complete lattices are special partial orders.

Properties of Complete Lattices

Lemma A.4.1.3 (Existence of Extremal Elements)

Let $(P, \sqsubseteq)$ be a complete lattice. Then there is

1. a least element in P, denoted by $\bot$, satisfying: $\bot = \bigsqcup \emptyset = \bigsqcap P$.
2. a greatest element in P, denoted by $\top$, satisfying: $\top = \bigsqcap \emptyset = \bigsqcup P$.

Lemma A.4.1.4 (Characterization Lemma)

Let $(P, \sqsubseteq)$ be a partial order. Then the following statements are equivalent:

1. $(P, \sqsubseteq)$ is a complete lattice.
2. Every subset of P has a least upper bound.
3. Every subset of P has a greatest lower bound.

Properties of Finite Lattices

Lemma A.4.1.5 (Finiteness implies Completeness)

If $(P, \sqsubseteq)$ is a finite lattice, then $(P, \sqsubseteq)$ is a complete lattice.

Corollary A.4.1.6 (Finiteness impl. Ex. of ext. Elem.)

If $(P, \sqsubseteq)$ is a finite lattice, then $(P, \sqsubseteq)$ has a least element and a greatest element.

Complete Semi-Lattices

Let $(P, \sqsubseteq)$ be a partial order, $P \neq \emptyset$.

Definition A.4.1.7 (Complete Semi-Lattice)

$(P, \sqsubseteq)$ is a complete

1. join semi-lattice (in German: Vereinigungshalbverband) iff $\forall \emptyset \neq S \subseteq P.\ \bigsqcup S \textit{ exists} \in P$.
2. meet semi-lattice (in German: Schnitthalbverband) iff $\forall \emptyset \neq S \subseteq P.\ \bigsqcap S \textit{ exists} \in P$.

Proposition A.4.1.8 (Extr. Bounds in C. Semi-Lat.)

If $(P, \sqsubseteq)$ is a complete

1. join semi-lattice, then $\bigsqcup P$ *exists* $\in P$ (whereas $\bigsqcup \emptyset$ ($\widehat{=} \bot$) does usually not exist in P).
2. meet semi-lattice, then $\bigsqcap P$ *exists* $\in P$ (whereas $\bigsqcap \emptyset$ ($\widehat{=} \top$) does usually not exist in P).

Informally: Least elements need not exist in complete join semi-lattices, greatest elements need not exist in complete meet semi-lattices.

Properties of Complete Semi-Lattices (2)

Lemma A.4.1.9 (Ex. great. El. in C. Join Semi-Lat.)

Let $(P, \sqsubseteq)$ be a complete join semi-lattice. Then:

$\bigsqcup P$ *exists* $\in P$ and is the (unique) greatest element in P that is usually denoted by $\top$, i.e., $\top = \bigsqcup P$.

Lemma A.4.1.10 (Ex. least El. in C. Meet Semi-Lat.)

Let $(P, \sqsubseteq)$ be a complete meet semi-lattice. Then:

$\bigsqcap P$ *exists* $\in P$ and is the (unique) least element in P that is usually denoted by $\bot$, i.e., $\bot = \bigsqcap P$.

Characterizing Upper and Lower Bounds (1)

...in complete semi-lattices.

Lemma A.4.1.11 (Char. u./l. Bounds in C. Semi-L.)

1. Let $(P, \sqsubseteq)$ be a complete join semi-lattice, and let $Q \subseteq P$ be a subset of P.

   If there is a lower bound for Q in P, i.e, if $\{p \in P \mid p \sqsubseteq Q\} \neq \emptyset$, then $\bigsqcap Q$ *exists* $\in P$ satisfying

   $$\bigsqcap Q = \bigsqcup \{p \in P \mid p \sqsubseteq Q\}$$

2. Let $(P, \sqsubseteq)$ be a complete meet semi-lattice, and let $Q \subseteq P$ be a subset of P.

   If there is an upper bound for Q in P, i.e, if $\{p \in P \mid Q \sqsubseteq p\} \neq \emptyset$, then $\bigsqcup Q$ *exists* $\in P$ satisfying

   $$\bigsqcup Q = \bigsqcap \{p \in P \mid Q \sqsubseteq p\}$$

Characterizing Upper and Lower Bounds (2)

Lemma A.4.1.12 (L./gr. Elements in C. Semi-L.)

If $(P, \sqsubseteq)$ is a complete

1. join semi-lattice and $\bigsqcup \emptyset$ *exists* $\in P$, then $\bigsqcup \emptyset$ is the (unique) least element in P, denoted by $\bot$, i.e., $\bot = \bigsqcup \emptyset$.
2. meet semi-lattice and $\bigsqcap \emptyset$ *exists* $\in P$, then $\bigsqcap \emptyset$ is the (unique) greatest element in P, denoted by $\top$, i.e., $\top = \bigsqcap \emptyset$.

Relating Complete Semi-Lattices and Lattices

Lemma A.4.1.13 (Complete Semi-Lattices&Lattices)

If $(P, \sqsubseteq)$ is a complete

1. join semi-lattice and $\bigsqcup \emptyset$ *exists* $\in P$
2. meet semi-lattice and $\bigsqcap \emptyset$ *exists* $\in P$

then $(P, \sqsubseteq)$ is a complete lattice.

Exercise A.4.1.14

Prove or disprove:

If $(P, \sqsubseteq)$ is a complete lattice, then

1. $(P \backslash \{\bot\}, \sqsubseteq_{\backslash \bot})$ is a complete join semi-lattice.
2. $(P \backslash \{\top\}, \sqsubseteq_{\backslash \top})$ is a complete meet semi-lattice.

where $\sqsubseteq_{\backslash \bot}$ and $\sqsubseteq_{\backslash \top}$ denote the restrictions of $\sqsubseteq$ from P to $P \backslash \{\bot\}$ and $P \backslash \{\top\}$, respectively. Proof or counterexample.

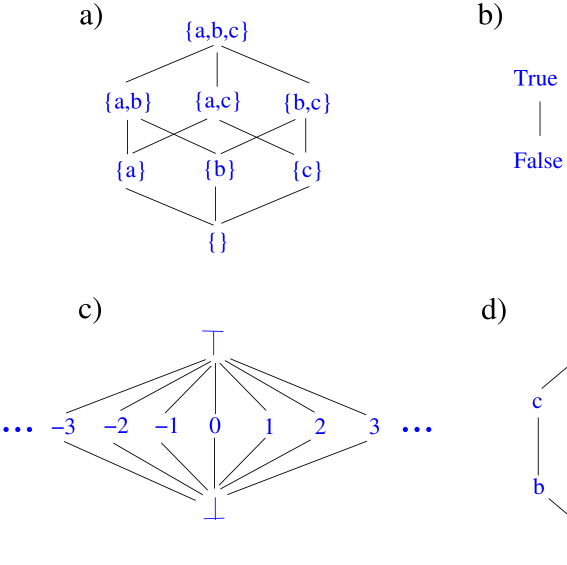
Relating Lattices and Complete Partial Orders

Lemma A.4.1.15 (Complete Lattices and CPOs)
If $(P, \sqsubseteq)$ is a complete lattice, then $(P, \sqsubseteq)$ is a CPO (i.e., a CCPO and DCPO).

Corollary A.4.1.16 (Finite Lattices and CPOs)
If $(P, \sqsubseteq)$ is a finite lattice, then $(P, \sqsubseteq)$ is a CPO (i.e., a CCPO and DCPO).

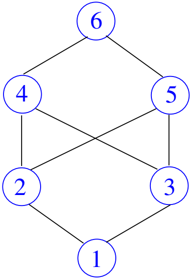
Note: Lemma A.4.1.15 does not hold for lattices.

Examples of Complete Lattices



(Anti-) Examples

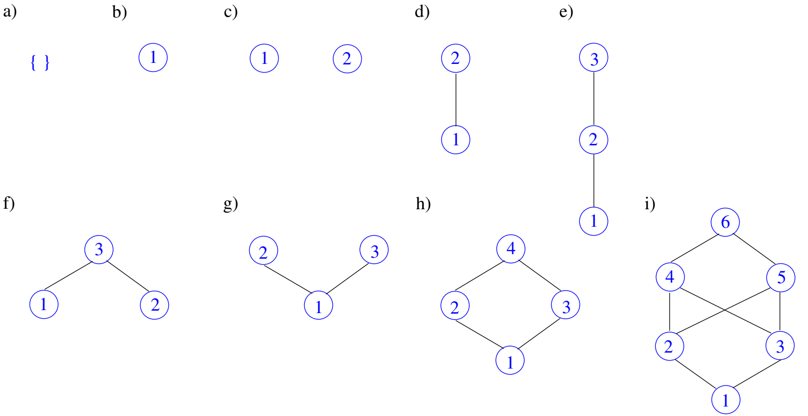
- The partial order $(P, \sqsubseteq)$ given by the below Hasse diagram is not a lattice (whereas it is a CPO).



- $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a complete lattice (and not a CPO).

Exercise A.4.1.17

Which of the partial orders given by the below Hasse diagrams are lattices? Which ones are complete lattices?



Exercise A.4.1.18

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \,|\, 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. lattice
2. complete lattice
3. complete join semi-lattice
4. complete meet semi-lattice

Proof or counterexample.

Summary, Overview

Corollary A.4.1.19

Let $P \neq \emptyset$ be a non-empty set, and $\sqsubseteq$ a relation on P. Then:

$$
\begin{array}{rcl}
 & & (P, \sqsubseteq) \text{ finite lattice (L. A.4.1.5)} \vee \\
 & & (P, \sqsubseteq) \text{ complete join semi-lattice and} \\
 & & \qquad \bigsqcup \emptyset \textit{ exists} \in P \text{ (L. A.4.1.13(1))} \vee \\
 & & (P, \sqsubseteq) \text{ complete meet semi-lattice and} \\
 & & \qquad \bigsqcap \emptyset \textit{ exists} \in P \text{ (L. A.4.1.13(2))} \\
 & \Rightarrow & (P, \sqsubseteq) \text{ complete lattice} \\
 \text{(D. A.4.1.2 and} & & \\
 \text{L. A.4.1.14)} & \Rightarrow & (P, \sqsubseteq) \text{ lattice and complete partial order} \\
 \text{(D. A.4.1.1 and} & & \\
 \text{D. A.3.1.1/2)} & \Rightarrow & (P, \sqsubseteq) \text{ partial order} \\
 \text{(D. A.2.1.2)} & \Rightarrow & (P, \sqsubseteq) \text{ pre-order}
\end{array}
$$

Excercise A.4.1.20

Let

$$\mathcal{QO}, \mathcal{PO}, \mathcal{L}, \mathcal{CPO}, \mathcal{CL}, \mathcal{FL}, \mathcal{CJSL}, \mathcal{CJSL}_\perp, \mathcal{CMSL}, \mathcal{CMSL}^\top$$

denote the sets of all quasi-orders $\mathcal{QO}$, partial orders $\mathcal{PO}$, lattices $\mathcal{L}$, complete partial orders $\mathcal{CPO}$, complete lattices $\mathcal{CL}$, finite lattices $\mathcal{FL}$, complete join semi-lattices without/with least element $\mathcal{CJSL}/\mathcal{CJSL}_\perp$, and meet semi-lattices without/with greatest element $\mathcal{CMSL}/\mathcal{CMSL}^\top$.

1. What further implications or equivalences hold in addition to those listed in Corollary A.4.1.19? (Proof or counterexample)
2. What inclusions or (set) equalities hold among $\mathcal{QO}$, $\mathcal{PO}$, $\mathcal{L}$, etc.? (Proof or counterexample)

A.4.2

Distributive, Additive Maps on Lattices

Distributive, Additive Maps on Lattices

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f \in [P \to P]$ be a map on P.

Definition A.4.2.1 (Distributive, Additive Map)

f is called

- distributive (or $\sqcap$-continuous) iff
$$\forall\, \emptyset \neq P' \subseteq P.\ f(\bigsqcap P') = \bigsqcap f(P')$$
(Preservation of greatest lower bounds)
- additive (or $\sqcup$-continuous) iff
$$\forall\, \emptyset \neq P' \subseteq P.\ f(\bigsqcup P') = \bigsqcup f(P')$$
(Preservation of least upper bounds)

Note: $\forall\, S \subseteq P.\ f(S) =_{df} \{\, f(s) \mid s \in S \}$

Characterizing Monotonicity

...in terms of the preservation of greatest lower and least upper bounds:

Lemma A.4.2.2 (Characterizing Monotonicity)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f \in [P \to P]$ be a map on P. Then:

$$\begin{aligned} f \text{ is monotonic } &\iff \forall\, P' \subseteq P.\ f(\bigsqcap P') \sqsubseteq \bigsqcap f(P') \\ &\iff \forall\, P' \subseteq P.\ f(\bigsqcup P') \sqsupseteq \bigsqcup f(P') \end{aligned}$$

Note: $\forall\, S \subseteq P.\ f(S) =_{df} \{\, f(s) \mid s \in S \}$

Useful Results on Mon., Distr., and Additivity

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f \in [P \to P]$ be a map on P.

Lemma A.4.2.3

f is distributive iff f is additive.

Lemma A.4.2.4

f is monotonic, if f is distributive (or additive).
(i.e., distributivity (or additivity) implies monotonicity)

A.4.3

Lattice Homomorphisms, Lattice Isomorphisms

Lattice Homomorphisms, Lattice Isomorphisms

Let $(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ be two lattices, and let $f \in [P \to R]$ be a map from P to R.

Definition A.4.3.1 (Lattice Homorphism)

f is called a lattice homomorphism, if

$$\forall\, p, q \in P.\ f(p \sqcup_P q) = f(p) \sqcup_Q f(q) \ \wedge$$
$$f(p \sqcap_P q) = f(p) \sqcap_Q f(q)$$

Definition A.4.3.2 (Lattice Isomorphism)

1. f is called a lattice isomorphism, if f is a lattice homomorphism and bijective.
2. $(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ are called isomorphic, if there is lattice isomorphism between P and R.

Useful Results (1)

Let $(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ be two lattices, and let $f \in [P \to R]$ be a map from P to R.

Lemma A.4.3.3

$$f \in [P \stackrel{hom}{\to} R] \;\Rightarrow\; f \in [P \stackrel{mon}{\to} R]$$

The reverse implication of Lemma A.4.3.3 does not hold, however, the following weaker relation holds:

Lemma A.4.3.4

$$\begin{aligned} f \in [P \stackrel{mon}{\to} R] \;\Rightarrow\; & \\ & \forall\, p, q \in P.\; f(p \sqcup_P q) \sqsupseteq_Q f(p) \sqcup_Q f(q) \;\wedge \\ & \qquad\qquad\quad f(p \sqcap_P q) \sqsubseteq_Q f(p) \sqcap_Q f(q) \end{aligned}$$

Useful Results (2)

Let $(P, \sqsubseteq_P)$ and $(R, \sqsubseteq_R)$ be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R.

Lemma A.4.3.5

$$f \in [P \overset{iso}{\rightarrow} R] \;\Rightarrow\; f^{-1} \in [R \overset{iso}{\rightarrow} P]$$

Lemma A.4.3.6

$$f \in [P \overset{iso}{\rightarrow} R] \iff f \in [P \overset{po-hom}{\rightarrow} R] \text{ wrt } \sqsubseteq_P \text{ and } \sqsubseteq_Q$$

A.4.4

Modular, Distributive, and Boolean Lattices

Modular Lattices

Let $(P, \sqsubseteq)$ be a lattice with meet operation $\sqcap$ and join operation $\sqcup$.

Lemma A.4.4.1

$$\forall\, p, q, r \in P.\ p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) \sqsubseteq (p \sqcup q) \sqcap r$$

Definition A.4.4.2 (Modular Lattice)

$(P, \sqsubseteq)$ is called modular, if

$$\forall\, p, q, r \in P.\ p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap r$$

Characterizing Modular Lattices

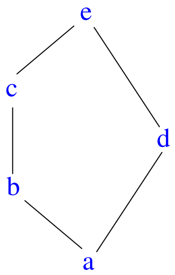
Theorem A.4.4.3 (Characterizing Modular Lattices)

A lattice $(P, \sqsubseteq)$ is

1. modular iff

$$\forall\, p, q, r \in P.\ p \sqsubseteq q,\ p \sqcap r = q \sqcap r,\ p \sqcup r = q \sqcup r \Rightarrow p = q$$

2. not modular iff $(P, \sqsubseteq)$ contains a sublattice, which is isomorphic to the lattice:



Distributive Lattices

Let $(P, \sqsubseteq)$ be a lattice with meet operation $\sqcap$ and join operation $\sqcup$.

Lemma A.4.4.4

1. $\forall\, p, q, r \in P.\ p \sqcup (q \sqcap r) \sqsubseteq (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall\, p, q, r \in P.\ p \sqcap (q \sqcup r) \sqsupseteq (p \sqcap q) \sqcup (p \sqcap r)$

Definition A.4.4.5 (Distributive Lattice)

$(P, \sqsubseteq)$ is called distributive, if

1. $\forall\, p, q, r \in P.\ p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall\, p, q, r \in P.\ p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$

Towards Characterizing Distributive Lattices

Lemma A.4.4.6

The following two statements are equivalent:

1. $\forall\, p, q, r \in P.\ p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall\, p, q, r \in P.\ p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$

Hence, it is sufficient to require the validity of property (1) or of property (2) in Definition A.4.4.5.

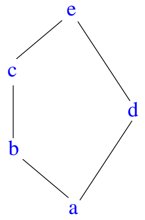
Characterizing Distributive Lattices

Let $(P, \sqsubseteq)$ be a lattice.

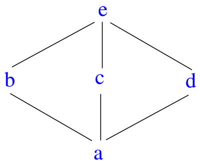
Theorem A.4.4.7 (Characterizing Distributive Lat.)

$(P, \sqsubseteq)$ is not distributive iff $(P, \sqsubseteq)$ contains a sublattice that is isomorphic to one of the below two lattices:

a)



b)



Corollary A.4.4.8

If $(P, \sqsubseteq)$ is distributive, then $(P, \sqsubseteq)$ is modular.

Boolean Lattices

Let $(P, \sqsubseteq)$ be a lattice with meet operation $\sqcap$, join operation $\sqcup$, least element $\bot$, and greatest element $\top$.

Definition A.4.4.9 (Complement)

Let $p, q \in P$. Then:

1. q is called a complement of p, if $p \sqcup q = \top$ and $p \sqcap q = \bot$.
2. P is called complementary, if every element in P has a complement.

Definition A.4.4.10 (Boolean Lattice)

$(P, \sqsubseteq)$ is called Boolean, if it is complementary, distributive, and $\bot \neq \top$.

Note: If $(P, \sqsubseteq)$ is Boolean, then every element $p \in P$ has an unambiguous unique complement in P, which is denoted by $\bar{p}$.

Useful Result

Lemma A.4.4.11

Let $(P, \sqsubseteq)$ be a Boolean lattice, and let $p, q, r \in P$. Then:

1. $\bar{\bar{p}} = p$ (Involution Law)
2. $\overline{p \sqcup q} = \bar{p} \sqcap \bar{q}, \quad \overline{p \sqcap q} = \bar{p} \sqcup \bar{q}$ (De Morgan Laws)
3. $p \sqsubseteq q \iff \bar{p} \sqcup q = \top \iff p \sqcap \bar{q} = \bot$
4. $p \sqsubseteq q \sqcup r \iff p \sqcap \bar{q} \sqsubseteq r \iff \bar{q} \sqsubseteq \bar{p} \sqcup r$

Boolean Lat. Homomorphisms/Isomorphisms

Let $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$ be two Boolean lattices, and let $f \in [P \to Q]$ be a map from P to Q.

Definition A.4.4.12 (Boolean Lattice Homomorphism)

f is called a Boolean lattice homomorphism, if f is a lattice homomorphism and

$$\forall\, p \in P.\ f(\bar{p}) = \overline{f(p)}$$

Definition A.4.4.13 (Boolean Lattice Isomorphism)

f is called a Boolean lattice isomorphism, if f is a Boolean lattice homomorphism and bijective.

Useful Results

Let $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$ be two Boolean lattices, and let $f \in [P \stackrel{bhom}{\to} Q]$ be a Boolean lattice homomorphism from P to Q.

Lemma A.4.4.14

$$f(\bot) = \bot \wedge f(\top) = \top$$

Lemma A.4.4.15

f is a Boolean lattice isomorphism iff $f(\bot) = \bot \wedge f(\top) = \top$

Summary, Overview

Corollary A.4.4.16

Let $P \neq \emptyset$ be a non-empty set, and $\sqsubseteq$ a relation on P. Then:

$$
\begin{array}{rcl}
 & & (P, \sqsubseteq) \text{ Boolean lattice} \\
(\text{Def. A.4.4.10}) & \Rightarrow & (P, \sqsubseteq) \text{ distributive lattice} \\
(\text{Cor. A.4.4.8}) & \Rightarrow & (P, \sqsubseteq) \text{ modular lattice} \\
(\text{Def. A.4.4.2}) & \Rightarrow & (P, \sqsubseteq) \text{ lattice} \\
(\text{Def. A.4.1.1}) & \Rightarrow & (P, \sqsubseteq) \text{ partial order} \\
(\text{Def. A.2.1.2}) & \Rightarrow & (P, \sqsubseteq) \text{ pre-order}
\end{array}
$$

Corollary A.4.4.17

$$\mathcal{QO} \supset \mathcal{PO} \supset \mathcal{L} \supset \mathcal{ML} \supset \mathcal{DL} \supset \mathcal{BL}$$

where all inclusions are proper and $\mathcal{QO}$, $\mathcal{PO}$, $\mathcal{L}$, $\mathcal{ML}$, $\mathcal{DL}$, and $\mathcal{BL}$ denote the sets of all quasi- (or pre-) orders, partial orders, lattices, modular, distributive, and Boolean lattices.

Exercise A.4.4.18

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \,|\, 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. modular lattice
2. distributive lattice
3. Boolean lattice

Proof or counterexample.

A.4.5

Mechanisms for Constructing Lattices

Common Lattice Constructions: Flat Lattices

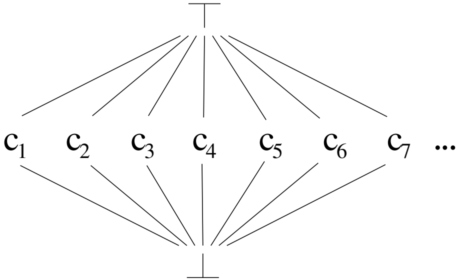
Lemma A.4.5.1 (Flat Lattice Construction)

Let C be a set. Then:

$(C \,\dot\cup\, \{\bot, \top\}, \sqsubseteq_{flat})$ with $\sqsubseteq_{flat}$ defined by

$$\forall\, c, d \in C \,\dot\cup\, \{\bot, \top\}.\ c \sqsubseteq_{flat} d \iff_{df} c = \bot \vee c = d \vee d = \top$$

is a complete lattice, a so-called flat lattice (or diamond lattice).



Lattice Constructions: Products, Sums,...

Like the principle for constructing flat CPOs also the principles for constructing

- non-strict products
- strict products
- separate sums
- coalesced sums
- continuous (here: additive, distributive) function spaces

carry over from CPOs to (complete) lattices (cf. App. A.3.3).

A.4.6

Order-theoretic and Algebraic View of Lattices

Motivation

In Definition A.4.1.1, we introduced lattices as special

- ordered sets $(P, \sqsubseteq)$

which induces an

- order-theoretic view of lattices.

Alternatively, lattices can be introduced as special

- algebraic structures $(P, \sqcap, \sqcup)$

which induces an

- algebraic view of lattices.

Next, we will show that both views are equivalent:

- Order-theoretically defined lattices can be considered algebraically and vice versa.

Lattices as Algebraic Structures

Definition A.4.6.1 (Algebraic Lattice)

An algebraic lattice is an algebraic structure $(P, \sqcap, \sqcup)$, where

- $P \neq \emptyset$ is a non-empty set.
- $\sqcap, \sqcup : P \times P \to P$ are two maps such that for all elements $p, q, r \in P$ the following laws hold (infix notation):
  - Commutative Laws: $p \sqcap q = q \sqcap p$
    $p \sqcup q = q \sqcup p$
  - Associative Laws: $(p \sqcap q) \sqcap r = p \sqcap (q \sqcap r)$
    $(p \sqcup q) \sqcup r = p \sqcup (q \sqcup r)$
  - Absorption Laws: $(p \sqcap q) \sqcup p = p$
    $(p \sqcup q) \sqcap p = p$

Properties of Algebraic Lattices

Let $(P, \sqcap, \sqcup)$ be an algebraic lattice.

Lemma A.4.6.2 (Idempotency Laws)

For all $p \in P$, the maps $\sqcap, \sqcup : P \times P \to P$ satisfy the following laws:

- Idempotency Laws: $p \sqcap p = p$
  $p \sqcup p = p$

Lemma A.4.6.3

For all $p, q \in P$, the maps $\sqcap, \sqcup : P \times P \to P$ satisfy:

1. $p \sqcap q = p \iff p \sqcup q = q$
2. $p \sqcap q = p \sqcup q \iff p = q$

Induced (Partial) Order

Let $(P, \sqcap, \sqcup)$ be an algebraic lattice.

Lemma A.4.6.4

The relation $\sqsubseteq \, \subseteq P \times P$ on P defined by

$$\forall \, p, q \in P. \; p \sqsubseteq q \iff_{df} p \sqcap q = p$$

is a partial order relation on P, i.e., $\sqsubseteq$ is reflexive, transitive, and antisymmetric.

Definition A.4.6.5 (Induced Partial Order)

The relation $\sqsubseteq$ defined in Lemma A.4.6.4 is called the induced (partial) order of $(P, \sqcap, \sqcup)$.

Properties of the Induced Partial Order

Let $(P, \sqcap, \sqcup)$ be an algebraic lattice, and let $\sqsubseteq$ be the induced partial order of $(P, \sqcap, \sqcup)$.

Lemma A.4.6.6

For all $p, q \in P$, the infimum ($\widehat{=}$ greatest lower bound) and the supremum ($\widehat{=}$ least upper bound) of the set $\{p, q\}$ exist and are given by the images of $\sqcap$ and $\sqcup$ applied to p and q, respectively, i.e.:

$$\forall\, p, q \in P.\ \bigsqcap\{p, q\} = p \sqcap q \ \wedge\ \bigsqcup\{p, q\} = p \sqcup q$$

Lemma A.4.6.6 can inductively be extended yielding:

Lemma A.4.6.7

Let $\emptyset \neq Q \subseteq P$ be a non-empty finite subset of P. Then:

$$\exists\, glb, lub \in P.\ glb = \bigsqcap Q \ \wedge\ lub = \bigsqcup Q$$

Algebraic Lattices Order-theoretically

Corollary A.4.6.8 (From $(P, \sqcap, \sqcup)$ to $(P, \sqsubseteq)$)

Let $(P, \sqcap, \sqcup)$ be an algebraic lattice. Then:

$(P, \sqsubseteq)$, where $\sqsubseteq$ is the induced partial order of $(P, \sqcap, \sqcup)$, is an order-theoretic lattice in the sense of Definition A.4.1.1.

Induced Algebraic Maps

Let $(P, \sqsubseteq)$ be an order-theoretic lattice.

Definition A.4.6.9 (Induced Algebraic Maps)

The partial order $\sqsubseteq$ of $(P, \sqsubseteq)$ induces two maps $\sqcap$ and $\sqcup$ from $P \times P$ to P defined by:

1. $\forall\, p, q \in P.\ p \sqcap q =_{df} \bigsqcap\{p, q\}$
2. $\forall\, p, q \in P.\ p \sqcup q =_{df} \bigsqcup\{p, q\}$

Properties of the Induced Algebraic Maps (1)

Let $(P, \sqsubseteq)$ be an order-theoretic lattice, and let $\sqcap$ and $\sqcup$ be the induced algebraic maps of $(P, \sqsubseteq)$.

Lemma A.4.6.10

Let $p, q \in P$. Then the following statements are equivalent:

1. $p \sqsubseteq q$
2. $p \sqcap q = p$
3. $p \sqcup q = q$

Properties of the Induced Algebraic Maps (2)

Let $(P, \sqsubseteq)$ be an order-theoretic lattice, and let $\sqcap$ and $\sqcup$ be the induced algebraic maps of $(P, \sqsubseteq)$.

Lemma A.4.6.11

For all $p, q, r \in P$, the induced maps $\sqcap$ and $\sqcup$ satisfy the following laws:

- Commutative Laws: $p \sqcap q = q \sqcap p$
  $p \sqcup q = q \sqcup p$
- Associative Laws: $(p \sqcap q) \sqcap r = p \sqcap (q \sqcap r)$
  $(p \sqcup q) \sqcup r = p \sqcup (q \sqcup r)$
- Absorption Laws: $(p \sqcap q) \sqcup p = p$
  $(p \sqcup q) \sqcap p = p$
- Idempotency Laws: $p \sqcap p = p$
  $p \sqcup p = p$

Corollary A.4.6.12 (From $(P, \sqsubseteq)$ to $(P, \sqcap, \sqcup)$)

Let $(P, \sqsubseteq)$ be an order-theoretic lattice. Then:

$(P, \sqcap, \sqcup)$, where $\sqcap$ and $\sqcup$ are the induced maps of $(P, \sqsubseteq)$, is an algebraic lattice in the sense of Definition A.4.6.1.

Equivalence (1)

...of the order-theoretic and the algebraic view of lattices.

From order-theoretic to algebraic lattices:

- An order-theoretic lattice $(P, \sqsubseteq)$ can be considered algebraically by switching from $(P, \sqsubseteq)$ to $(P, \sqcap, \sqcup)$, where $\sqcap$ and $\sqcup$ are the induced maps of $(P, \sqsubseteq)$.

From algebraic to order-theoretic lattices:

- An algebraic lattice $(P, \sqcap, \sqcup)$ can be considered order-theoretically by switching from $(P, \sqcap, \sqcup)$ to $(P, \sqsubseteq)$, where $\sqsubseteq$ is the induced partial order of $(P, \sqcap, \sqcup)$.

Equivalence (2)

Together, this allows us to simply speak of a lattice P, and to speak only more precisely of P as an

- order-theoretic lattice $(P, \sqsubseteq)$
- algebraic lattice $(P, \sqcap, \sqcup)$

if we want to emphasize that we think of P as a special ordered set or as a special algebraic structure.

Bottom and Top vs. Zero and One (1)

Let P be a lattice with a least and a greatest element.

Considering P

- order-theoretically as $(P, \sqsubseteq)$, it is appropriate to think of its least and greatest element in terms of bottom $\perp$ and top $\top$ with
  - Bottom $\perp \in P$: $\perp = \bigsqcup \emptyset$
  - Top $\top \in P$: $\top = \bigsqcap \emptyset$
- algebraically as $(P, \sqcap, \sqcup)$, it is appropriate to think of its least and greatest element in terms of Zero $\mathbf{0}$ and One $\mathbf{1}$, where $(P, \sqcap, \sqcup)$ is said to have a (if existent, uniquely determined)
  - Zero $\mathbf{0} \in P$: $\forall\, p \in P.\ p \sqcup \mathbf{0} = p$
  - One $\mathbf{1} \in P$: $\forall\, p \in P.\ p \sqcap \mathbf{1} = p$

Bottom and Top vs. Zero and One (2)

Lemma A.4.6.13

Let P be a lattice. Then:

- $(P, \sqsubseteq)$ has a bottom element $\bot$ iff $(P, \sqcap, \sqcup)$ has a zero element $\mathbf{0}$, and in that case:

$$(\bigsqcup \emptyset =)\ \bot = \mathbf{0}$$

- $(P, \sqsubseteq)$ has a top element $\top$ iff $(P, \sqcap, \sqcup)$ has a one element $\mathbf{1}$, and in that case:

$$(\bigsqcap \emptyset =)\ \top = \mathbf{1}$$

On the Adequacy of the two Lattice Views

In mathematics, usually the

- algebraic view of a lattice is more appropriate as it is in line with other algebraic structures ('a set together with some maps satisfying a number of laws'), e.g., groups, rings, fields, vector spaces, categories, etc., which are investigated and dealt with in mathematics.

In computer science, usually the

- order-theoretic view of a lattice is more appropriate, since the order relation can often be interpreted and understood as '$\cdot$ carries more/less information than $\cdot$,' '$\cdot$ is more/less defined than $\cdot$,' '$\cdot$ is stronger/weaker than $\cdot$,' etc., which often fits naturally to problems investigated and dealt with in computer science.

Exercise A.4.6.14

Let $(\mathbb{N}_0, \sqsubseteq)$ be the lattice with $\sqsubseteq =_{df} \mid$, where $\mid$ denotes the divisibility relation on the natural numbers $\mathbb{N}_0$, i.e., the relation '$\cdot$ divides $\cdot$' (w/out remainder), e.g. $5 \mid 35$.

Provide the definition of $(\mathbb{N}_0, \wedge, \vee)$, i.e., write down the algebraically defined counterpart of $(\mathbb{N}_0, \sqsubseteq)$. To this end, provide the definition of the meet and join operation on $\mathbb{N}_0 \times \mathbb{N}_0$:

1. $\wedge : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
2. $\vee : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

What is the

1. zero element $\mathbf{0}$
2. one element $\mathbf{1}$

of $(\mathbb{N}_0, \wedge, \vee)$?

A.5

Fixed Point Theorems

A.5.1

Fixed Points, Towers

Fixed Points of Functions

Definition A.5.1.1 (Fixed Point)

Let M be a set, let $f \in [M \to M]$ be a function on M, and let $m \in M$ be an element of M. Then:

m is called a fixed point of f iff $f(m) = m$.

Least, Greatest Fixed Points in Partial Orders

Definition A.5.1.2 (Least, Greatest Fixed Point)

Let $(P, \sqsubseteq)$ be a partial order, let $f \in [P \to P]$ be a function on P, and let p be a fixed point of f, i.e., $f(p) = p$. Then:

p is called the

- least fixed point of f, denoted by μf, iff $\forall\, q \in P.\ f(q) = q \Rightarrow p \sqsubseteq q$
- greatest fixed point of f, denoted by νf, iff $\forall\, q \in P.\ f(q) = q \Rightarrow q \sqsubseteq p$

Towers in Chain Complete Partial Orders

Definition A.5.1.3 (f-Tower in C)

Let $(C, \sqsubseteq)$ be a CCPO, let $f \in [C \to C]$ be a function on C, and let $T \subseteq C$ be a subset of C. Then:

T is called an f-tower in C iff

1. $\perp \in T$.
2. If $t \in T$, then also $f(t) \in T$.
3. If $T' \subseteq T$ is a chain in C, then $\bigsqcup T' \in T$.

Least Towers in Chain Complete Partial Orders

Lemma A.5.1.4 (The Least f-Tower in C)

The intersection

$$I =_{df} \bigcap \{ T \mid T \ f\text{-tower in } C \}$$

of all f-towers in C is the least f-tower in C, i.e.,

1. I is an f-tower in C.
2. $\forall \, T \ f$-tower in C. $I \subseteq T$.

Lemma A.5.1.5 (Least f-Towers and Chains)

The least f-tower in C is a chain in C, if f is expanding.

A.5.2

Fixed Point Theorems for Complete Partial Orders

Fixed Points of Exp./Monotonic Functions

Fixed Point Theorem A.5.2.1 (Expanding Function)

Let $(C, \sqsubseteq)$ be a CCPO, and let $f \in [C \stackrel{exp}{\to} C]$ be an expanding function on C. Then:

The supremum of the least f-tower in C is a fixed point of f.

Fixed Point Theorem A.5.2.2 (Monotonic Function)

Let $(C, \sqsubseteq)$ be a CCPO, and let $f \in [C \stackrel{mon}{\to} C]$ be a monotonic function on C. Then:

f has a unique least fixed point μf, which is given by the supremum of the least f-tower in C.

Note

- Theorem A.5.2.1 and Theorem A.5.2.2 ensure the existence of a fixed point for expanding functions and of a unique least fixed point for monotonic functions, respectively, but do not provide constructive procedures for computing or approximating them.
- This is in contrast to Theorem A.5.2.3, which does so for continuous functions. In practice, continuous functions are thus more important and considered where possible.

Least Fixed Points of Continuous Functions

Fixed Point Theorem A.5.2.3 (Knaster,Tarski,Kleene)

Let $(C, \sqsubseteq)$ be a CCPO, and let $f \in [C \stackrel{con}{\to} C]$ be a continuous function on C. Then:

f has a unique least fixed point $\mu f \in C$, which is given by the supremum of the (so-called) Kleene chain $\{\perp, f(\perp), f^2(\perp), \ldots\}$, i.e.:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{\perp, f(\perp), f^2(\perp), \ldots\}$$

Note: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

Proof of Fixed Point Theorem A.5.2.3 (1)

We have to prove:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup \{ f^i(\bot) \mid i \geq 0 \}$$

1. exists,
2. is a fixed point of f,
3. is the least fixed point of f.

Proof of Fixed Point Theorem A.5.2.3 (2)

1. Existence

- By definition of $\perp$ as the least element of C and of f^0 as the identity on C we have: $\perp = f^0(\perp) \sqsubseteq f^1(\perp) = f(\perp)$.
- Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
  $\forall\, i,j \in \mathbb{N}_0.\ i < j \Rightarrow f^i(\perp) \sqsubseteq f^{i+1}(\perp) \sqsubseteq f^j(\perp)$.
- Hence, the set $\{f^i(\perp) \mid i \geq 0\}$ is a (possibly infinite) chain in C.
- Since $(C, \sqsubseteq)$ is a CCPO and $\{f^i(\perp) \mid i \geq 0\}$ a chain in C, this implies by definition of a CCPO that the least upper bound of the chain $\{f^i(\perp) \mid i \geq 0\}$

$$\bigsqcup \{f^i(\perp) \mid i \geq 0\} = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \text{ exists.}$$

Proof of Fixed Point Theorem A.5.2.3 (3)

2. Fixed point property

$$
\begin{array}{rcl}
 & & f(\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)) \\
(f \text{ continuous}) & = & \bigsqcup_{i \in \mathbb{N}_0} f(f^i(\perp)) \\
 & = & \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp) \\
(C' =_{df} \{f^i \perp \mid i \geq 1\} \text{ is a chain } \Rightarrow & & \\
\bigsqcup C' \text{ exists } = \perp \sqcup \bigsqcup C') & = & \perp \sqcup \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp) \\
(f^0(\perp) =_{df} \perp) & = & \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)
\end{array}
$$

Proof of Fixed Point Theorem A.5.2.3 (4)

3. Least fixed point property

- Let c be an arbitrary fixed point of f. Then: $\perp \sqsubseteq c$.
- Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
  $\forall\, i \in \mathbb{N}_0.\ f^i(\perp) \sqsubseteq f^i(c)\ (= c)$.
- Since c is a fixed point of f, this implies:
  $\forall\, i \in \mathbb{N}_0.\ f^i(\perp) \sqsubseteq c\ (= f^i(c))$.
- Thus, c is an upper bound of the set $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$.
- Since $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$ is a chain, and $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is by definition the least upper bound of this chain, we obtain the desired inclusion

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c.$$

□

Least Conditional Fixed Points

Let $(C, \sqsubseteq)$ be a CCPO, let $f \in [C \to C]$ be a function on C, and let $d, c_d \in C$ be elements of C.

Definition A.5.2.4 (Least Conditional Fixed Point)

c_d is called the least conditional fixed point of f wrt d (in German: kleinster bedingter Fixpunkt) iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e.:

$$\forall x \in C.\ f(x) = x \ \wedge\ d \sqsubseteq x \Rightarrow c_d \sqsubseteq x$$

Least Cond. Fixed Points of Cont. Functions

Theorem A.5.2.5 (Conditional Fixed Point Theorem)

Let $(C, \sqsubseteq)$ be a CCPO, let $d \in C$, and let $f \in [C \stackrel{con}{\to} C]$ be a continuous function on C which is expanding for d, i.e., $d \sqsubseteq f(d)$. Then:

f has a least conditional fixed point $\mu f_d \in C$, which is given by the supremum of the (generalized) Kleene chain $\{d, f(d), f^2(d), \ldots\}$, i.e.:

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}$$

Finite Fixed Points

Let $(C, \sqsubseteq)$ be a CCPO, let $d \in C$, and let $f \in [C \overset{mon}{\to} C]$ be a monotonic function on C.

Theorem A.5.2.6 (Finite Fixed Point Theorem)

If two succeeding elements in the Kleene chain of f are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(\perp) = f^{i+1}(\perp)$, then we have: $\mu f = f^i(\perp)$.

Theorem A.5.2.7 (Finite Conditional FP Theorem)

If f is expanding for d, i.e., $d \sqsubseteq f(d)$, and two succeeding elements in the (generalized) Kleene chain of f wrt d are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(d) = f^{i+1}(d)$, then we have: $\mu f_d = f^i(d)$.

Note: Theorems A.5.2.6 and A.5.2.7 do not require continuity of f. Monotonicity (and expandingness) of f suffice(s).

Towards the Existence of Finite Fixed Points

Let $(P, \sqsubseteq)$ be a partial order, and let $p, r \in P$.

Definition A.5.2.8 (Chain-finite Partial Order)

$(P, \sqsubseteq)$ is called chain-finite (in German: kettenendlich) iff P does not contain an infinite chain.

Definition A.5.2.9 (Finite Element)

p is called

- finite iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ does not contain an infinite chain.
- finite relative to r iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ does not contain an infinite chain.

Existence of Finite Fixed Points

...there are numerous sufficient conditions ensuring the existence of a least finite fixed point of a function f, which often hold in practice (cf. Nielson/Nielson 1992), e.g.:

- the domain or the range of f are finite or chain-finite,
- the least fixed point of f is finite,
- f is of the form $f(c) = c \sqcup g(c)$ with g a monotonic function on a chain-finite (data) domain.

Fixed Point Theorems, Lattices, and DCPOs

Note: Complete lattices (cf. Lemma A.4.1.13) and DCPOs with a least element (cf. Lemma A.3.1.5) are CCPOs, too.

Thus, we can conclude:

Corollary A.5.2.10 (Fixed Points, Lattices, DCPOs)

The fixed point theorems of Chapter A.5.2 hold for functions on complete lattices and on DCPOs with a least element, too.

A.5.3

Fixed Point Theorems for Lattices

Fixed Points of Monotonic Functions

Fixed Point Theorem A.5.3.1 (Knaster, Tarski)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f \in [P \stackrel{mon}{\to} P]$ be a monotonic function on P. Then:

1. f has a unique least fixed point $\mu f \in P$, which is given by $\mu f = \bigsqcap\{p \in P \mid f(p) \sqsubseteq p\}$.
2. f has a unique greatest fixed point $\nu f \in P$, which is given by $\nu f = \bigsqcup\{p \in P \mid p \sqsubseteq f(p)\}$.

Characterization Theorem A.5.3.2 (Davis)

Let $(P, \sqsubseteq)$ be a lattice. Then:

$(P, \sqsubseteq)$ is complete iff every $f \in [P \stackrel{mon}{\to} P]$ has a fixed point.

The Fixed Point Lattice of Mon. Functions

Theorem A.5.3.3 (Lattice of Fixed Points)

Let $(P, \sqsubseteq)$ be a complete lattice, let $f \in [P \stackrel{mon}{\to} P]$ be a monotonic function on P, and let $Fix(f) =_{df} \{p \in P \mid f(p) = p\}$ be the set of all fixed points of f. Then:

Every subset $F \subseteq Fix(f)$ has a supremum and an infimum in $Fix(f)$, i.e., $(Fix(f), \sqsubseteq_{\mid Fix(f)})$ is a complete lattice.

Theorem A.5.3.4 (Ordering of Fixed Points)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f \in [P \stackrel{mon}{\to} P]$ be a monotonic function on P. Then:

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) \sqsubseteq \mu f \sqsubseteq \nu f \sqsubseteq \bigsqcap_{i \in \mathbb{N}_0} f^i(\top)$$

Fixed Points of Add./Distributive Functions

For additive and distributive functions, the leftmost and the rightmost inequality of Theorem A.5.3.4 become equalities:

Fixed Point Theorem A.5.3.5 (Knaster,Tarski,Kleene)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f \in [P \to P]$ be a function on P. Then: f has a unique

1. least fixed point $\mu f \in P$ given by $\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)$, if f is additive, i.e., $f \in [P \stackrel{add}{\to} P]$.
2. greatest fixed point $\nu f \in P$ given by $\nu f = \bigsqcap_{i \in \mathbb{N}_0} f^i(\top)$, if f is distributive, i.e., $f \in [P \stackrel{dis}{\to} P]$.

Recall: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

A.6
Fixed Point Induction

Admissible Predicates

Fixed point induction allows proving properties of fixed points. Essential is the notion of admissible predicates:

Definition A.6.1 (Admissible Predicate)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $\phi : P \to \mathbb{B}$ be a predicate on P. Then:

ϕ is called admissible (or $\sqcup$-admissible) iff for every chain $C \subseteq P$ holds:

$$(\forall\, c \in C.\ \phi(c)) \Rightarrow \phi(\bigsqcup C)$$

Lemma A.6.2

Let $(P, \sqsubseteq)$ be a complete lattice, and let $\phi : P \to \mathbb{B}$ be an admissible predicate on P. Then: $\phi(\bot) = \mathit{true}$.

Proof. The admissibility of ϕ implies $\phi(\bigsqcup \emptyset) = \mathit{true}$. Moreover, we have $\bot = \bigsqcup \emptyset$, which completes the proof.

Sufficient Conditions for Admissibility

Theorem A.6.3 (Admissibility Condition 1)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $\phi : P \to \mathbb{B}$ be a predicate on P. Then:

ϕ is admissible, if there is a complete lattice $(Q, \sqsubseteq_Q)$ and two additive functions $f, g \in [P \stackrel{add}{\to} Q]$, such that

$$\forall\, p \in P.\ \phi(p) \iff f(p) \sqsubseteq_Q g(p)$$

Theorem A.6.4 (Admissibility Condition 2)

Let $(P, \sqsubseteq)$ be a complete lattice, and let $\phi, \psi : P \to \mathbb{B}$ be two admissible predicates on P. Then:

The conjunction of ϕ and ψ, the predicate $\phi \;\wedge\; \psi$ defined by

$$\forall\, p \in P.\ (\phi \;\wedge\; \psi)(p) =_{df} \phi(p) \wedge \psi(p)$$

is admissible.

Fixed Point Induction on Complete Lattices

Theorem A.6.5 (Fixed Point Induction on C. Lat.)

Let $(P, \sqsubseteq)$ be a complete lattice, let $f \in [P \stackrel{add}{\to} P]$ be an additive function on P, and let $\phi : P \to \mathbb{B}$ be an admissible predicate on P. Then:

The validity of

- $\forall\, p \in P.\ \phi(p) \Rightarrow \phi(f(p))$ (Induction step)

implies the validity of $\phi(\mu f)$.

Note: The induction base, i.e., the validity of $\phi(\bot)$, is implied by the admissibility of ϕ (cf. Lemma A.6.2) and proved when verifying the admissibility of ϕ.

Fixed Point Induction on CCPOs

The notion of admissibility of a predicate carries over from complete lattices to CCPOs.

Theorem A.6.6 (Fixed Point Induction on CCPOs)

Let $(C, \sqsubseteq)$ be a CCPO, let $f \in [C \overset{mon}{\to} C]$ be a monotonic function on C, and let $\phi : C \to \mathbb{B}$ be an admissible predicate on C. Then:

The validity of

- $\forall\, c \in C.\ \phi(c) \Rightarrow \phi(f(c))$ (Induction step)

implies the validity of $\phi(\mu f)$.

Note: Theorem A.6.6 holds (of course still), if we replace the CCPO $(C, \sqsubseteq)$ by a complete lattice $(P, \sqsubseteq)$.

A.7

Completions, Embeddings

A.7.1

Downsets

Downsets

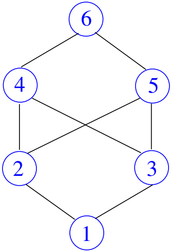
Definition A.7.1.1 (Downset)

Let $(P, \sqsubseteq)$ be a partial order, let $D \subseteq P$ be a subset of P, and let $p, q \in P$ with $p \sqsubseteq q$. Then:

1. D is called a downset (or lower set or order ideal) (in German: Abwärtsmenge) of P, if: $q \in D \Rightarrow p \in D$.
2. $\mathcal{D}(P)$ denotes the set of all downsets of P.

Example

Let $(P, \sqsubseteq)$ be the partial order given by the below Hasse diagram.



Then, e.g.:

1. $\emptyset, P \in \mathcal{D}(P)$, $\forall\, q \in P.\ \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)$
2. $\{1,3\}, \{1,2,3\}, \{1,2,3,4\} \in \mathcal{D}(P)$
3. $\{2,3\}, \{2,4,5\}, \{1,2,4,5\} \notin \mathcal{D}(P)$

Properties of Downsets

Lemma A.7.1.2

Let $(P, \sqsubseteq)$ be a partial order, let $q \in P$, and $Q \subseteq P$. Then:

1. $\emptyset \in \mathcal{D}(P)$, $P \in \mathcal{D}(P)$, are (trivial) downsets of P.
2. $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)$.
3. $\downarrow Q =_{df} \{p \in P \mid \exists\, q \in Q.\ p \sqsubseteq q\} \in \mathcal{D}(P)$.
4. $Q \in \mathcal{D}(P) \iff Q = \downarrow Q$

Lemma A.7.1.3

Let $(P, \sqsubseteq)$ be a partial order, and let $p, q \in P$. Then the following statements are equivalent:

1. $p \sqsubseteq q$
2. $\downarrow p \subseteq \downarrow q$
3. $\forall\, D \in \mathcal{D}(P).\ q \in D \Rightarrow p \in D$.

Characterization of Downsets

Lemma A.7.1.4 (Downsets of a PO)

Let $(P, \sqsubseteq)$ be a partial order. Then:

$$\mathcal{D}(P) = \{\downarrow Q \mid Q \subseteq P\}$$

Corollary A.7.1.5

Let $(P, \sqsubseteq)$ be a partial order, let $D \in \mathcal{D}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in D \Rightarrow p \in D$.

The Lattice of Downsets: Complete & Distr.

Let $(P, \sqsubseteq)$ be a partial order, let $\mathcal{D}(P)$ be the set of downsets of P, and let $\subseteq$ denote set inclusion.

Theorem A.7.1.6 (Complete & Distr. L. of Downsets)

$(\mathcal{D}(P), \subseteq)$ is a complete and distributive lattice, the so-called downset lattice of P, with set intersection $\cap$ as meet operation, set union $\cup$ as join operation, least element $\emptyset$, and greatest element P.

Recall: Complete lattices are CCPOs and DCPOs, too (cf. Lemma A.4.1.13). Thus, we have:

Corollary A.7.1.7 (The CCPO/DCPO of Downsets)

$(\mathcal{D}(P), \subseteq)$ is a CCPO and a DCPO with least element $\emptyset$.

From POs to Lattices, CCPOs, and DCPOs

Construction Principle:

Theorem A.7.1.6 and Corollary A.7.1.7 yield a construction principle that shows how to construct

- a complete lattice and thus also a CCPO and a DCPO

from a given partial order $(P, \sqsubseteq)$ (cf. Appendix A.3.3 and Appendix A.4.5).

Principal Downsets

The downsets of the form $\{p \in P \mid p \sqsubseteq q\}$ of a partial order $(P, \sqsubseteq)$ considered in Lemma A.7.1.2(2) are peculiar, and will reoccur as so-called principal ideals (cf. Chapter A.7.2) and principal cuts (cf. Chapter A.7.3) of lattices. Therefore, we introduce these distinguished downsets explicitly.

Definition A.7.1.8 (Principal Downsets of a PO)

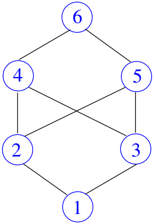
Let $(P, \sqsubseteq)$ be a partial order, and let $q \in P$ be an element of P. Then:

1. $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\}$ denotes the principal downset (in German: Hauptabwärtsmenge) generated by q.
2. $\mathcal{PD}(P) = \{\downarrow q \mid q \in P\}$ denotes the set of all principal downsets of P.

Downsets, Directed Sets (1)

...principal downsets of partial orders are directed but usually not strongly directed.

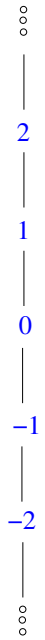
Example 1: Consider the below partial order $(P, \sqsubseteq)$:



- $\forall\, p \in P.\ \downarrow p =_{df} \{r \mid r \sqsubseteq p\}$ *directed* $\in \mathcal{D}(P)$.
- $\forall\, p \in P \backslash \{6\}.\ \downarrow p$ *strongly directed* $\in \mathcal{D}(P)$.
- $\downarrow 6 =_{df} \{r \mid r \sqsubseteq 6\} = \{1,2,3,4,5,6\} = P \in \mathcal{D}(P)$ is a downset of P, however, it is not strongly directed, since its subsets $\{2,3\}, \{1,2,3\} \subseteq \downarrow 6$ do not have a least upper bound in $\downarrow 6 = P$ (though upper bounds: $4, 5, 6$).

Downsets, Directed Sets (2)

Example 2: Consider the below lattice $(\mathbb{Z}, \leq)$:

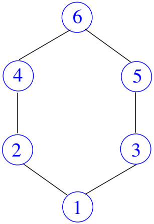


- $\mathcal{D}(\mathbb{Z}) = \emptyset \cup \mathcal{PD}(\mathbb{Z}) \cup \mathbb{Z} =$
  $\emptyset \cup \{\downarrow z =_{df} \{r \in \mathbb{Z} \mid r \leq z\} \mid z \in \mathbb{Z}\} \cup \mathbb{Z}$
- $\forall\, S \in \mathcal{D}(\mathbb{Z}).$ S *directed* but *not strongly directed* (since it lacks a least element).

Downsets, Directed Sets (3)

...arbitrary downsets even of complete lattices are usually not strongly directed, though directed.

Example 3: Consider the below complete lattice $(P, \sqsubseteq)$:



- E.g., the downsets
  - $\downarrow\{4,5\} =_{df} \{r \mid r \sqsubseteq 4 \vee r \sqsubseteq 5\}\} = \{1,2,3,4,5\} \in \mathcal{D}(P)$
  - $\downarrow\{3,4\} =_{df} \{r \mid r \sqsubseteq 3 \vee r \sqsubseteq 4\}\} = \{1,2,3,4\} \in \mathcal{D}(P)$

  of P are directed but not strongly directed: The subsets $\{2,3\} \subseteq \downarrow\{4,5\}$ and $\{1,2,3\} \subseteq \downarrow\{3,4\}$ do not have a least upper bound in $\downarrow\{4,5\}$ and $\downarrow\{3,4\}$, respectively.

A.7.2

Ideal Completion: Embedding of Lattices

Lattice Ideals

Definition A.7.2.1 (Lattice Ideal)

Let $(P, \sqsubseteq)$ be a lattice, let $\emptyset \neq I \subseteq P$ be a non-empty subset of P, and let $p, q \in P$. Then:

1. I is called an ideal (or lattice ideal) of P, if:
   - $p, q \in I \Rightarrow p \sqcup q \in I$.
   - $q \in I \Rightarrow p \sqcap q \in I$.
2. $\mathcal{I}(P)$ denotes the set of all ideals of P.

Properties of Lattice Ideals

Lemma A.7.2.2 (Ideal Properties 1)

Let $(P, \sqsubseteq)$ be a lattice, let $I \in \mathcal{I}(P)$, and let $q \in I$. Then:

1. $\{p \in P \mid p \sqsubseteq q\} \subseteq I$.
2. $P \in \mathcal{I}(P)$ is a (trivial) ideal of P.

Lemma A.7.2.3 (Ideal Properties 2)

Let $(P, \sqsubseteq)$ be a lattice with least element $\perp$, and $I \in \mathcal{I}(P)$. Then:

1. $\perp \in I$.
2. $\{\perp\} \in \mathcal{I}(P)$ is a (trivial) ideal of P.

Characterizing Lattice Ideals

Theorem A.7.2.4 (Ideal Characterization)

Let $(P, \sqsubseteq)$ be a lattice, and let $\emptyset \neq I \subseteq P$ be a non-empty subset of P. Then:

$$I \in \mathcal{I}(P) \;\text{ iff }\; \forall\, p, q \in P.\; p, q \in I \iff p \sqcup q \in I$$

Lattice Ideals and Order Ideals

Lemma A.7.2.5

Let $(P, \sqsubseteq)$ be a lattice, let $I \in \mathcal{I}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in I \Rightarrow p \in I$.

Corollary A.7.1.5 – recalled

Let $(P, \sqsubseteq)$ be a partial order, let $D \in \mathcal{D}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in D \Rightarrow p \in D$.

Corollary A.7.2.6

Let $(P, \sqsubseteq)$ be a lattice, and let $I \subseteq P$. Then:

$$I \in \mathcal{I}(P) \Rightarrow I \in \mathcal{D}(P) \quad \text{(i.e., } \mathcal{I}(P) \subseteq \mathcal{D}(P)\text{)}.$$

Note: The reverse implication of Corollary A.7.2.6 does not hold.

The Complete Lattice of Ideals

Theorem A.7.2.7 (The Complete Lattice of Ideals)

Let $(P, \sqsubseteq)$ be a lattice with least element $\perp$, and let $\sqsubseteq_{\mathcal{I}}$ be the following ordering relation on the set $\mathcal{I}(P)$ of ideals of P:

$$\forall\, I, J \in \mathcal{I}(P).\ I \sqsubseteq_{\mathcal{I}} J \text{ iff } I \subseteq J$$

Then: $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ is a complete lattice, the so-called lattice of ideals of P, with join operation $\sqcup_{\mathcal{I}}$ defined by

$$\forall\, I, J \in \mathcal{I}(P).\ I \sqcup_{\mathcal{I}} J =_{df} \{p \in P \mid \exists\, i \in I,\ j \in J.\ p \sqsubseteq i \sqcup j\}$$

and meet operation $\sqcap_{\mathcal{I}}$ defined by

$$\forall\, I, J \in \mathcal{I}(P).\ I \sqcap_{\mathcal{I}} J =_{df} I \cap J$$

and with least element $\{\perp\}$ and greatest element P.

Principal Ideals

Lemma A.7.2.8

Let $(P, \sqsubseteq)$ be a lattice, and let $q \in P$ be an element of P. Then:

$$\downarrow q = \{p \in P \mid p \sqsubseteq q\} \text{ ideal} \in \mathcal{I}(P).$$

Definition A.7.2.9 (Principal Ideal)

Let $(P, \sqsubseteq)$ be a lattice, and let $q \in P$ be an element of P. Then:

1. $\downarrow q$ is called the principal ideal of P generated by q.
2. $\mathcal{PI}(P) =_{df} \{\downarrow q \mid q \in P\}$ denotes the set of all principal ideals of P.

Towards the Sublattice of Principal Ideals

Lemma A.7.2.10

Let $(P, \sqsubseteq)$ be a lattice with least element, and let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of ideals of P. Then:

$$\forall\, q, r \in P.\ \downarrow q \sqcap_{\mathcal{I}} \downarrow r = \downarrow(q \sqcap r) \wedge\ \downarrow q \sqcup_{\mathcal{I}} \downarrow r = \downarrow(q \sqcup r)$$

The Sublattice of Principal Ideals

Theorem A.7.2.11 (Sublattice of Principal Ideals)

Let $(P, \sqsubseteq)$ be a lattice with least element, let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of ideals of P, let $\mathcal{PI}(P)$ be the set of the principal ideals of P, and let $\sqsubseteq_{\mathcal{PI}}$ be the restriction of $\sqsubseteq_{\mathcal{I}}$ onto $\mathcal{PI}(P)$. Then:

$(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ is a sublattice of $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$.

Note: The sublattice $(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ of $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ is

- usually not complete, not even if $(P, \sqsubseteq)$ is complete.

  (The lattice $(\mathbb{Z}, \leq)$, e.g., enriched with a least element $\perp$ and a greatest element $\top$ is complete, while the lattice of its principal ideals $(\mathcal{PI}(\mathbb{Z}), \subseteq_{\mathcal{PI}})$ is not.)

Ideal Completion and Embedding of a Lattice

Theorem A.7.2.12 (Ideal Completion & Embedding)

Let $(P, \sqsubseteq)$ be a lattice with least element, and let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of its ideals. Then:

The map

$$e_{\mathcal{I}} : P \to \mathcal{PI}(P) \text{ defined by } \forall\, p \in P.\ e_{\mathcal{I}}(p) =_{df} \downarrow p$$

is a lattice isomorphism between P and the (sub)lattice $\mathcal{PI}(P)$ of its principal ideals.

Intuitively

Theorem A.7.2.12 shows how a lattice $(P, \sqsubseteq)$ with least element

- can be considered a sublattice of the complete lattice of the ideals of P; in more detail, how it can be considered the sublattice $(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ of the complete lattice $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$.

A.7.3

Cut Completion: Embedding of Partial Orders and Lattices

Cuts

Definition A.7.3.1 (Cut)

Let $(P, \sqsubseteq)$ be a partial order, and let $Q \subseteq P$ be a subset of P. Then:

1. Q is called a cut (in German: Schnitt) of P, if $Q = LB(UB(Q))$.
2. $\mathcal{C}(P)$ denotes the set of all cuts of P.

Properties of Cuts

Lemma A.7.3.2

Let $(P, \sqsubseteq)$ be a partial order, and let $q \in P$ be an element of P. Then:

1. $LB(\{q\}) =_{df} \downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\} \in \mathcal{C}(P)$
2. $LB(UB(\{q\}) = \{p \in P \mid p \sqsubseteq q\} = LB(\{q\})$

Note: If $(P, \sqsubseteq)$ is a lattice,

1. Lemma A.7.3.2(1) yields that principal ideals are cuts of P:
$$\forall\, q \in P.\ \langle q \rangle =_{df} \{p \in P \mid p \sqsubseteq q\} = LB(\{q\}) \in \mathcal{C}(P)$$
$$(\text{or: } \forall\, Q \subseteq P.\ Q \in \mathcal{PI}(P) \Rightarrow Q \in \mathcal{C}(P))$$
2. Lemma A.7.3.2(2) characterizes the principal ideals of P in terms of the function composition $LB \circ UB$.

Principal Cuts

Definition A.7.3.3 (Principal Cut)

Let $(P, \sqsubseteq)$ be a partial order, and let $q \in P$ be an element of P. Then:

1. $\downarrow q =_{df} LB(UB(\{q\}))$ is called the principal cut of P generated by q.
2. $\mathcal{PC}(P) =_{df} \{\downarrow q \mid q \in P\}$ denotes the set of all principal cuts of P.

Properties of Cuts and Ideals of Lattices

Lemma A.7.3.4

Let $(P, \sqsubseteq)$ be a lattice with least element, and let $Q \subseteq P$. Then:

$$Q \in \mathcal{C}(P) \Rightarrow Q \in \mathcal{I}(P)$$

Corollary A.7.3.5

Let $(P, \sqsubseteq)$ be a lattice with least element, and let $Q \subseteq P$. Then:

$$Q \in \mathcal{C}(P) \Rightarrow Q \neq \emptyset$$

Note: Corollary A.7.3.5 does not hold for partial orders.

The Complete Lattice of Cuts

Theorem A.7.3.6 (The Complete Lattice of Cuts)

Let $(P, \sqsubseteq)$ be a partial order, and let $\sqsubseteq_{\mathcal{C}}$ be the following ordering relation on the set $\mathcal{C}(P)$ of cuts of P:

$$\forall\, C, D \in \mathcal{C}(P).\ C \sqsubseteq_{\mathcal{C}} D \text{ iff } C \subseteq D$$

Then: $(\mathcal{C}(P), \sqsubseteq_{\mathcal{C}})$ is a complete lattice, the so-called lattice of cuts of P, with join operation $\sqcup_{\mathcal{C}}$ defined by

$$\forall\, C, D \in \mathcal{C}(P).\ C \sqcup_{\mathcal{C}} D =_{df} \bigcap \{E \in \mathcal{C}(P) \mid C \cup D \subseteq E\}$$

and meet operation $\sqcap_{\mathcal{C}}$ defined by

$$\forall\, C, D \in \mathcal{C}(P).\ C \sqcap_{\mathcal{C}} D =_{df} C \cap D$$

and with least element $\{\bot\}$ and greatest element P.

Cut Completion and Embedding of a PO

Theorem A.7.3.7 (PO Cut Completion & Embedd'g)

Let $(P, \sqsubseteq)$ be a partial order, and let $(\mathcal{C}(P), \sqsubseteq_{\mathcal{C}})$ be the complete lattice of its cuts. Then:

The map

$e_{\mathcal{C}} : P \to \mathcal{PC}(P)$ defined by $\forall\, p \in P.\ e_{\mathcal{C}}(p) =_{df} LB(UB(\{p\}))$

is an order isomorphism between P and the partial order $(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ of the principal cuts of P.

Cut Completion and Embedding of a Lattice

Theorem A.7.3.8 (Lattice Cut Completion & Emb'g)

Let $(P, \sqsubseteq)$ be a lattice, let $(\mathcal{C}(P), \sqsubseteq_{\mathcal{C}})$ be the complete lattice of its cuts, and let $e_{\mathcal{C}} : P \to \mathcal{PC}(P)$ be the map of Theorem A.7.3.7. Then:

$(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ is a sublattice of $(\mathcal{C}(P), \sqsubseteq)$ and $e_{\mathcal{C}}$ is a lattice isomorphism between P and the sublattice $\mathcal{PC}(P)$ of the principal cuts of P.

A.7.4

Downset Completion: Embedding of Partial Orders

Downsets, Ideals, and Cuts

Lemma A.7.4.1

We have:

1. $\mathcal{C}(P) \subseteq \mathcal{D}(P)$, if $(P, \sqsubseteq)$ is a partial order.
2. $\mathcal{C}(P) \subseteq \mathcal{I}(P) \subseteq \mathcal{D}(P)$, if $(P, \sqsubseteq)$ is a lattice with least element.

Downset Completion and Embedding of a PO

Theorem A.7.4.2 (Downset Completion and Emb.'g)

Let $(P, \sqsubseteq)$ be a partial order, and let $(\mathcal{D}(P), \subseteq)$ be the complete and distributive lattice of its downsets (cf. Theorem A.7.1.6). Then:

The map $e_{\mathcal{C}} : P \to \mathcal{PC}(P)$ (of Theorem A.7.3.7) defined by

$$\forall\, p \in P.\ e_{\mathcal{C}}(p) =_{df} LB(UB(\{p\}))$$

is an order isomorphism between P and the partial order $(\mathcal{PC}(P), \subseteq)$ of the principal cuts of P, or, equivalently, the map $e_{\mathcal{C}} : P \to \mathcal{D}(P)$ defined as above is a partial order embedding of $(\mathcal{PC}(P), \subseteq)$ into $(\mathcal{D}(P), \subseteq)$.

Intuitively

Theorem A.7.4.2 shows how a partial order $(P, \sqsubseteq)$

- can be considered a partial order of the complete and distributive lattice of its downsets; in more detail, how it can be considered the partial order $(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ of the complete and distributive lattice $(\mathcal{D}(P), \sqsubseteq_{\mathcal{D}})$.

A.7.5

Application: Lists and Streams

Technically

...the construction of Chapter A.7.4 works by

- switching from the elements p of a set P partially ordered by a relation $\sqsubseteq$ to the principal downsets $\downarrow p \in \mathcal{PD}(P)$ of the set of downsets $\mathcal{D}(P)$ of P ordered by the subset inclusion $\subseteq$.

Identifying

- every element $p \in P$ with its principal downset

$$\downarrow p =_{df} \{r \mid r \sqsubseteq p\} \in \mathcal{PD}(P)$$

yields an

- embedding of P into $\mathcal{PD}(P) =_{df} \{\downarrow q \mid q \in P\}$, i.e., a function $e : P \to \mathcal{PD}(P)$ with

$$\forall\, p, q \in P.\ p \sqsubseteq q \;\Leftrightarrow \downarrow p \subseteq \downarrow q$$

From Monotonic to Continuous Functions

...completion is the key to Theorem A.7.5.1:

Let $(P, \sqsubseteq_P)$ be a partial order, let $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\}$ for $q \in P$, let $\mathcal{PD}(P) =_{df} \{\downarrow q \mid q \in P\}$, and let $(C, \sqsubseteq_C)$ be a CPO.

Theorem A.7.5.1 (From Monotonicity to Continuity)

A monotonic function $f \in [P \stackrel{mon}{\to} C]$ can uniquely be extended to a continuous function $\hat{f} \in [\mathcal{PD}(P) \stackrel{con}{\to} C]$.

Application: Lists and Streams (1)

Lemma A.7.5.2 (The CPO of Lists and Streams)

Let L be the set of all finite and infinite lists, and let $\sqsubseteq_{pfx}$ be the prefix relation '$\cdot$ is a prefix of $\cdot$' on L defined by

$$\forall\, l, l'' \in L.\ l \sqsubseteq_{pfx} l'' \Longleftrightarrow_{df}$$
$$l = l'' \vee (l \textit{ finite } \wedge \exists\, l' \in L.\ l +\!\!+ l' = l'')$$

Then: $(L, \sqsubseteq_{pfx})$ is a CPO (i.e., a CCPO and DCPO).

Lemma A.7.5.3 (Downsets of the Set of Lists)

Let L be the set of all finite and infinite lists, and let $\mathcal{PD}(L) = \{\downarrow l \mid l \in L\}$ be the set of principal downsets of L. Then:

1. $\downarrow l =_{df} \{l' \in L \mid l' \sqsubseteq_{pfx} l\}$ is a directed set (even a strongly directed set), i.e., a directed downset of lists.
2. $(\mathcal{PD}(L), \subseteq)$ is a CPO (i.e., a CCPO and DCPO).

Application: Lists and Streams (2)

Putting these findings together, we obtain:

- The set of downsets of lists ordered by set inclusion is a CPO.
- Every (infinite) chain of ever longer finite lists represents the corresponding stream, the supremum of this chain.
- Theorem A.7.4.3 allows the application of a function to a stream to be approximated and computed by applying the function to the finite prefixes of the stream yielding a chain of approximations of the stream that would result from the application of the function to the stream itself.
- Continuity ensures the correctness of this procedure: it yields the equality of the supremum of the computed chain of approximations and the result of applying the continuous function to the argument stream itself.

Application: Lists and Streams (3)

Together, this implies:

- Recursive equations and functions on streams as considered in Chapter 2 are well defined.

A.8

References, Further Reading

Appendix A: Further Reading (1)

- André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
- Roland Backhouse, Roy Crole, Jeremy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002. (Chapter 1, Ordered Sets and Complete Lattices by Hilary A. Priestley; Chapter 2, Algebras and Coalgebras by Peter Aczel; Chapter 4, Calculating Functional Programs by Jeremy Gibbons)
- Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Vieweg+Teubner, 2008.

Appendix A: Further Reading (2)

- Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012. (Kapitel 1, Ordnungen und Verbände; Kapitel 2.4, Vollständige Verbände; Kapitel 3, Fixpunkttheorie mit Anwendungen; Kapitel 4, Vervollständigung und Darstellung mittels Vervollständigung; Kapitel 5, Wohlgeordnete Mengen und das Auswahlaxiom)

- Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013. (Kapitel 2, Verbände und Ordnungen; Kapitel 3.4, Vollständige Verbände; Kapitel 4, Fixpunkttheorie mit Anwendungen; Kapitel 5, Vervollständigung und Darstellung mittels Vervollständigung; Kapitel 6, Wohlgeordnete Mengen und das Auswahlaxiom)

Appendix A: Further Reading (3)

- Garret Birkhoff. *Applications of Lattice Algebra*. Mathematical Proceedings of the Cambridge Philosophical Society 30(2):115-122, 1934.
- Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.
- Peter Crawley, Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.
- Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002. (Chapter 1, Ordered Sets; Chapter 2, Lattices and Complete Lattices; Chapter 8, CPOs and Fixpoint Theorems)

Appendix A: Further Reading (4)

- Anne C. Davis. *A Characterization of Complete Lattices*. Pacific Journal of Mathematics 5(2):311-319, 1955.
- Marcel Erné. *Einführung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
- Helmuth Gericke. *Theorie der Verbände*. Bibliographisches Institut, 2. Auflage, 1967.
- George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 1998. (Chapter 1, First Concepts; Chapter 2, Distributive Lattices; Chapter 3, Congruences and Ideals; Chapter 5, Varieties of Lattices)
- George Grätzer. *Lattice Theory: Foundation*. Birkhäuser, 2011.

Appendix A: Further Reading (5)

- George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. I*. Birkhäuser, 2014.
- George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. II*. Birkhäuser, 2016.
- Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001. (Chapter 6, Ordered Pairs; Chapter 7, Relations; Chapter 8, Functions)
- Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.

Appendix A: Further Reading (6)

- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 11, Proof by Induction; Chapter 14.6, Inductive Properties of Infinite Lists)
- Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009. (Chapter 3, Functions, Sequences, and Relations)
- Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1952. (Reprint, North Holland, 1980)
- Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998. (Chapter 4, Functions; Chapter 6, Relations)

Appendix A: Further Reading (7)

- David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 1, Collecting Things Together: Sets; Chapter 2, Comparing Things: Relations)
- George Markowsky. *Chain-complete Posets and Directed Sets with Applications*. Algebra Universalis 6(1):53-68, 1976.
- Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 4, Denotational Semantics)

Appendix A: Further Reading (8)

- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 5, Denotational Semantics)

- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Appendix A, Partially Ordered Sets)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten; Kapitel 10.2, Ein bisschen Mathematik: CPOs und Fixpunkte)

- Steven Roman. *Lattices and Ordered Sets*. Springer-V., 2008.

Appendix A: Further Reading (9)

- Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 5.1, Ordnungsrelationen; Kapitel 5.2, Ordnungen und Teilstrukturen)
- Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Chapter 5.1, Order Relations; Chapter 5.2, Orders and Substructures)
- Alfred Tarski. *A Lattice-theoretical Fixpoint Theorem and its Applications*. Pacific Journal of Mathematics 5(2):285-309, 1955.

Appendix A: Further Reading (10)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about Programs; Chapter 17.9, Proof revisited)
- Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008. (Chapter 5, Fixed Points; Chapter 105, Software Testing; Chapter 106, Formal Methods; Chapter 107, Verification and Validation)