Fortgeschrittene funktionale Programmierung

LVA 185.A05, VU 2.0, ECTS 3.0
SS 2013

(Stand: 27.06.2013)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen

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)

Part VI: Extensions and Prospectives

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

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

- 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 making it clear what those advantages are."

Chapter 1.1

Setting the Stage

Introductory Statement

A matter of fact:

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

Claim:

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

Next:

- Providing evidence for this claim

Background

Functional programming owes its name to the facts that

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

Folk Knowledge: Soft Facts

...of characteristics & advantages of functional programming:

Functional programs are

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

$\Rightarrow$ 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

$\Rightarrow$ functional programs are thus easier to cope with mathematically (e.g. for proving their correctness)

Observation

...the commonly found previous list of characteristics and advantages of functional programming is

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

Folk Knowledge: Hard(er) Facts

Aren't there any hard(er) facts providing evidence for substantial and "real" advantages?

Yes, there are, e.g.:

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

Open Issue:

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

Hardly.

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

Summing up: Lesson learnt

- The "standard catalogue" is not satisfying
  - It does not provide any help in exploiting the power of functional languages
    - Programs cannot be written which are particularly lacking in assignment statements, or which are particularly referentially transparent
  - It does not provide a yardstick of program quality, thus no model to strive for
- We need a positive characterization of the vital nature of
  - functional programming, of its strengths
  - what makes a "good" functional program, of what a functional programmer should strive for

Towards a Positive Characterization

Structured vs. non-structured programming

...provides an analogue to compare with:

Structured programs are

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

⇒ easier to mathematically cope with than unstructured programs

Note: This is essentially a negative "is-not"-characterization, too.

Towards a Positive Characterization (Cont'd)

Conceptually more important:

Structured programs are:

- designed modularly in contrast to non-structured programs
- Structured programming is more efficient/productive for this reason
  - Small modules are easier and faster to write and to maintain
  - Re-use becomes easier
  - Modules can be tested independently

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

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

Thesis

- The expressiveness of a language that supports modular design depends much on the power of the concepts and primitives allowing to combine solutions of subproblems to the solution of the overall problem (keyword: glue; example: making of a chair)
- Functional programming provides two new, especially powerful glues:
  1. Higher-order functions (functionals)
  2. Lazy evaluation

  They offer conceptually new opportunities for modularization and re-use (beyond the more technical ones of lexical scoping, separate compilation, etc.), and make them more easily to achieve.
- Modularization (smaller, simpler, more general) is the guideline, which should be followed by functional programmers in the course of programming

In the following

- Glueing functions together

  ⇝ The clou: Higher-order functions

- Glueing programs together

  ⇝ The clou: Lazy evaluation

Chapter 1.2

Glueing Functions Together

Glueing Functions Together

Syntax (in the flavour of Miranda^{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 `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

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

2. Computing the product of the elements of a list

```
product = reduce multiply 1
                   where multiply 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
```

Intuition

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`

Examples:

```
reduce add 0:
  cons 1 (cons 2 (cons 3 nil))
    ->>  add 1 (add 2 (add 3 0))
    ->> 6

reduce multiply 1:
  cons 1 (cons 2 (cons 3 nil))
    ->> multiply 1 (multiply 2 (multiply 3 1))
    ->> 6
```

More Applications 1(5)

Observation: `reduce cons nil` copies a list of elements

This allows:

5. Concatenation of lists

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

Example:

```
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])
 ->> cons 1 (cons 2 (cons 3 (cons 4 nil)))
 ->> [1,2,3,4]
```

More Applications 2(5)

6. Doubling each element of a list

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

More Applications 3(5)

The function `doubleandcons` can be modularized further:

- First step

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

- Second step

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

where "." denotes the composition of functions:

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

Note: For checking correctness consider:

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

which yields as desired:

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

Putting things together, we obtain:

6a. Doubling each element of a list

```
doubleall = reduce (cons . double) nil
```

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

6b. Doubling each element of a list

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

where `map` applies any function `f` to all the elements of a list.

More Applications 5(5)

After these preparative steps it is just as well possible:

7. Adding the elements of a matrix

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

Homework: Think about how summatrix works.

Summing up

By decomposing (modularizing) and representing a simple function (`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 without any further programming effort!

Generalization

...to more complex data structures:

Trees:

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

Example:

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

Generalization (Cont'd)

Analogously to `reduce` on lists we introduce a functional `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`)

- The first one to replace `node` with
- The second one to replace `cons` with
- The third one to replace `nil` with

Applications 1(4)

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

Applications 2(4)

1. Adding the labels of the leaves of a tree

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

Example:

Using the tree introduced previously, we obtain:

```
add 1
    (add (add 2 0)
         (add (add 3
                   (add (add 4 0) 0))
              0))
->> 10
```

Applications 3(4)

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

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

Example:

```
cons 1
  (append (cons 2 nil)
          (append (cons 3
                         (append (cons 4 nil) nil))
                  nil))
->> [1,2,3,4]
```

Applications 4(4)

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

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

Summing up

- The elegance of the preceding examples is a consequence of combining
  - a higher-order function and
  - a specific specializing function
- Once the higher order function is implemented, lots of further functions can be implemented almost without any further effort!

Summing up (Cont'd)

- Lesson learnt: Whenever a new data type is introduced, implement first a higher-order function allowing to process values of this type (e.g., visiting each component of a structured data value such as nodes in a graph or tree).
- Benefits: Manipulating elements of this data type becomes easy; knowledge about this data type is locally concentrated and encapsulated.
- Look&feel: Whenever a new data structure demands a new control structure, then this control structure can easily be added following the methodology used above (to some extent this resembles the concepts known from conventional extensible languages).

Reminder to initial Thesis

- The expressiveness of a language that supports modular design depends much on the power of the concepts and primitives allowing to combine solutions of subproblems to the solution of the overall problem (keyword: glue; example: making of a chair).
- Functional programming provides two new, especially powerful glues:
  1. Higher-order functions (functionals)
  2. Lazy evaluation

  They offer conceptually new opportunities for modularization and re-use (beyond the more technical ones of lexical scoping, separate compilation, etc.), and make them more easily to achieve.
- Modularization (smaller, simpler, more general) is the guideline, which should be followed by functional programmers in the course of programming.

Reminder (Cont'd)

So far, we talked about:

- Higher-order functions as glue for glueing functions together

Next we will talk about:

- Lazy evaluation as glue for glueing programs together

Chapter 1.2

Glueing Programs Together

Glueing Programs Together

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

- If `f` and `g` are (such) programs, then also

  ```
  g . f
  ```

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

  ```
  g (f input)
  ```

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

Functional Glue

Lazy evaluation allows a more elegant approach:

- Decomposing a problem into a
  - generator
  - selector

  component, which are then glued together.

Intuition:

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

Example 1: Computing Square Roots

Computing Square Roots (according to Newton-Raphson)

Given: `N` Wanted: `squareRoot(N)`

Iteration formula:

```
a(n+1) = (a(n) + N/a(n)) / 2
```

Justification: If the approximations converge to some limit `a`, we have:

```
    a = (a + N/a) / 2
=> 2a = a + N/a
    a = N/a
    a*a = N
    a = squareRoot(N)
```

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

For later comparison we consider first

...a typical imperative (Fortran-) implementation:

```
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
```

$\rightsquigarrow$ essentially monolithic, not decomposable.

The Functional Version 1(4)

Computing the next approximation from the previous one:

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

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

```
[a0, f a0, f(f a0), f(f(f a0)),...]
```

The Functional Version 2(4)

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

Generator:

```
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
```

The Functional Version 3(4)

Note: The evaluation of

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

does not terminate!

Remedy: Computing `squareroot N` up to a given tolerance eps > 0. Crucial: The selector component implemented by:

Selector:

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

Still to do: Combining the components/modules:

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

$\rightsquigarrow$ We are done!

The Functional Version 4(4)

Summing up:

- `repeat`: generator component:
  `[a0, f a0, f(f a0), f(f(f a0)),...]`
  ...potentially infinite, no limit on the length.
- `within`: selector component:
  f^i a0 with abs(f^i a0 - f^{i+1} a0) <= eps
  ...lazy evaluation ensures that the selector function is applied eventually ⇒ termination!

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

Re-Use of Modules

Next, we want to provide evidence that

- generator
- selector

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

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

Evidence of Generator-Modularity

Consider a new criterion for termination:

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

New Selector:

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

Still to do: (Re-)combining the components/modules:

```
relativesqrt a0 eps N
  = relative eps (repeat (next N) a0)
```

⇝ We are done!

Note the Re-Use

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

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

Next, we want to re-use the module selector.

Example 2: Numerical Integration

Numerical Integration

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

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

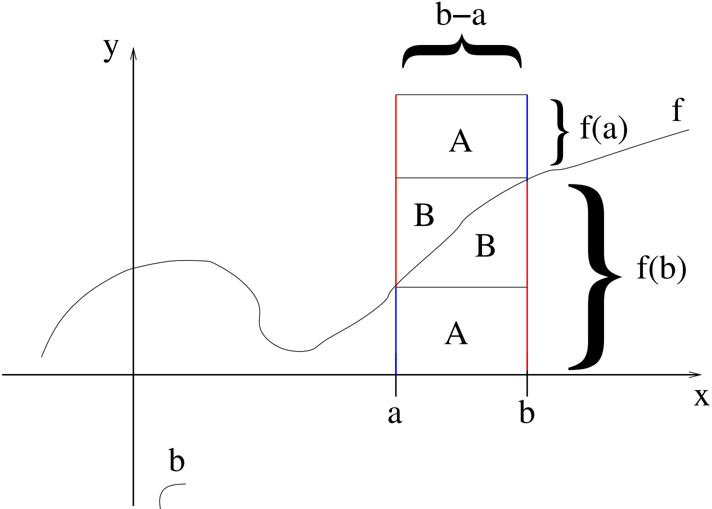
Naive Implementation:

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

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

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

Illustration



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

Refinements 1(4)

Idea

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

The function `integrate` implements this strategy:

Generator:

```
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
```

Reminder:

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

Refinements 2(4)

- `integrate` is sound but inefficient (many redundant computations of `f a`, `f b`, and `f mid`)

The following version of `integrate` is free of 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
```

Refinements 3(4)

Obviously, the evaluation of

```
integrate f a b
```

does not terminate!

Remedy: Computing `integrate f a b` up to some limit eps > 0.

Two Selectors:

```
Variant A:  within eps (integrate f a b)
Variant B:  relative eps (integrate f a b)
```

Refinements 4(4)

Summing up:

- Generator component:
  `integrate`
  ...potentially infinite, no limit on the length.
- Selector component:
  `within`, `relative`
  ...lazy evaluation ensures that the selector function is applied eventually ⇒ termination!

Note the Re-Use

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

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

Again, lazy evaluation is the key to synchronize the generator and selector module!

Example 3: Numerical Differentiation

Numerical Differentiation

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

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

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

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

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

Refinements

Generate a sequence of approximations getting successively "better":

Generator:

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

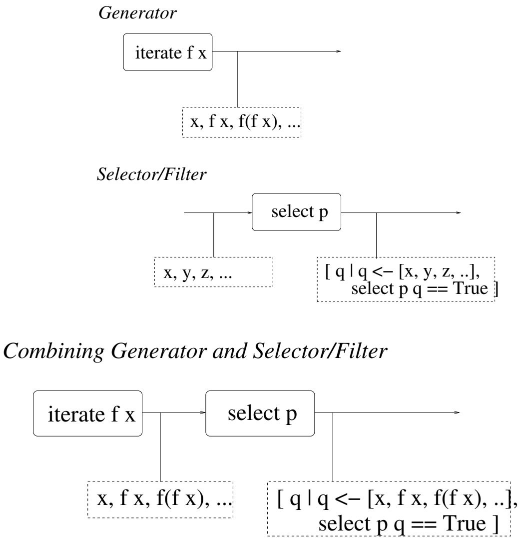
Select a sufficiently precise approximation:

Selector:

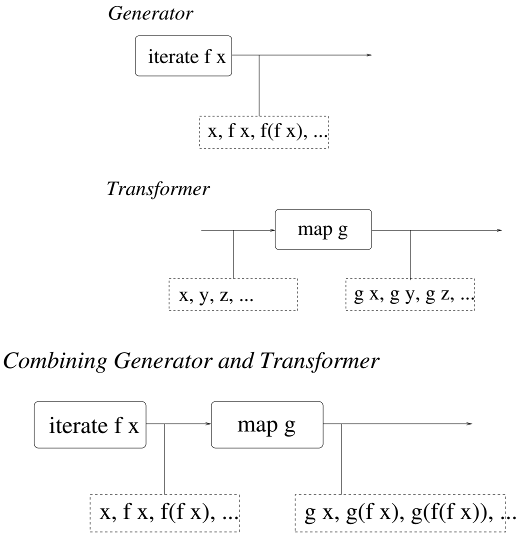
```
within esp (differentiate h0 f x)
```

Implementing the selector: Homework

The Generator/Selector Principle at a Glance



The Generator/Transformer Princ. at a Glance



Summary of Findings (1)

The composition pattern, which in fact is common to all three examples becomes again obvious. It consists of a

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

Summary of Findings (2)

Thesis

- Modularity is the key to programming in the large

Observation

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

Summary of Findings (3)

Facts

- Functional programming offers two new kinds of glue:
  - Higher-order functions (glueing functions)
  - Lazy evaluation (glueing programs)
- Higher-order functions and lazy evaluation allow substantially new exciting modular decompositions of problems (by offering elegant composition means) as here given evidence by an array of simple, yet impressive examples
- In essence, it is the superior glue, which makes functional programs to be written so concisely and elegantly (not the absence of assignments, etc.)

Summary of Findings (4)

Guidelines

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

  are the tools for achieving this!

Chapter 1.4

Summing Up

Summing Up: Lazy or Eager Evaluation

The final conclusion of John Hughes:

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

Outlook

John Hughes identifies

- higher-order functions
- lazy evaluation

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

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

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

Chapter 1: Further Reading (1)

- 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.
- 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.

Part II

Programming Principles

Chapter 2

Programming with Streams

Motivation

Streams = Infinite Lists

Programming with streams

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

    as instances of the Generator/Prune Paradigm
  - Pitfalls and remedies
- Foundations
  - Well-definedness
  - Proving properties of programs with streams

Chapter 2.1

Streams

Streams

Jargon

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

Streams

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

More on this in this chapter.

Streams

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

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

Convention

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

This is motivated by:

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

First Examples of Streams

- Built-in streams in Haskell

```
[2..]    ->> [2,3,4,5,6,7,...
[3,5..] ->> [3,5,7,9,11,...
```

- User-defined streams in Haskell

  The infinite lists of "twos"

  2,2,2,...

  In Haskell this can be realized:

  - using list comprehension: `[2,2..]`
  - (co-) recursively: `twos = 2 : twos`

    Illustration

```
twos ->> 2 : twos
     ->> 2 : 2 : twos
     ->> 2 : 2 : 2 : twos
     ->> ...
```

  twos represents an infinite list; synonymously, a stream.

Corecursive Definitions

- Definitions of the form

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

  defining the streams of "ones," "twos," and "threes" look like recursive definitions.

- However, they lack a base case.
- Definitions of the above form are called
  - corecursive
- Corecursive definitions always yield infinite objects.

More corecursively defined Streams

- The stream of natural numbers `nats`

```
nats = 0 : map (+1) nats
```

- The stream of even natural numbers `evens`

```
evens = 0 : map (+2) evens
```

- The stream of odd natural numbers `odds`

```
odds = 1 : map (+2) odds
```

More Streams

- The stream of natural numbers

```
theNats = 0 : zipWith (+) ones theNats
```

- The stream of powers of an integer

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

- The prelude function iterate

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

The function iterate generates the stream

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

Application: powers can be defined 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

evens  = iterate (+2) 0

odds   = iterate (+2) 1

powers = iterate (*n) 1
```

Functions on Streams

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

Application

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

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

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

Reminder

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

(Church/Rosser-Theorem)

- Normal-order reduction corresponds to leftmost-outermost evaluation

Recall: Let

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

Then, both in

- `ignore twos 42`
- `twos 'ignore' 42`

the leftmost-outermost operator is given by the call `ignore`.

Functions on Streams (Cont'd)

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

Application

```
addFirstTwo twos ->> 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
             ->> ...
```

Primes: The Sieve of Eratosthenes 1(3)

Intuition

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

Illustration

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

Primes: The Sieve of Eratosthenes 2(3)

The stream of primes:

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

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

Primes: The Sieve of Eratosthenes 3(3)

Illustration: By stepwise evaluation

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

Pitfalls in Applications

Implementing a prime number test (naively):

Let

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

Then

`member primes 7` ...yields "True" (as expected!)

But

`member primes 6` ...does not terminate!

Homework: Why fails the above implementation? How can `primes` be embedded into a calling context allowing us to decide if some argument is prime or not?

Random Numbers 1(2)

Generating a sequence of (pseudo-) random numbers:

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

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

Choosing

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

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

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

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

Random Numbers 2(2)

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

This can be achieved by scaling 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

```
scale 42.0 51.0 randomSequence
```

Principles of Modularization

...related to streams:

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

The Fibonacci Numbers 1(5)

The sequence of Fibonacci Numbers

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

is defined in terms of the function

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

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

The Fibonacci Numbers 2(5)

We have already learned that a naive implementation like

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

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

- inacceptably inefficient and slow!

The Fibonacci Numbers 3(5)

Illustration: By stepwise evaluation

```
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(5)

```
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(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!)

Reminder: Complexity 1(3)

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

Reminder: $\mathcal{O}$ Notation

- Let $f : \alpha \rightarrow IR^+$ 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:

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

Reminder: Complexity 2(3)

Examples of typical cost functions:

Code	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
$\mathcal{O}(n\ log\ n)$	"*n log n*"	...tenthousandfold effort
$\mathcal{O}(n^2)$	quadratic	...millionfold effort
$\mathcal{O}(n^3)$	cubic	...billiardfold effort
$\mathcal{O}(n^c)$	polynomial	... gigantic much effort (for big c)
$\mathcal{O}(2^n)$	exponential	...hopeless

Reminder: Complexity 3(3)

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

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...

Remedy

- Streams can (often) help!

Fibonacci Numbers Reconsidered 1(2)

Idea

```
0  1  1  2  3  5  8  13...  Sequence of Fib. Numbers
1  1  2  3  5  8  13 21...  Remainder of the S. of F. N.
-----------------------------------------------------------
1  2  3  5  8  13 21 34...  Remain. of the rem. of the
                                 sequ. of Fibonacci Numbers
```

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

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

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

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

Fibonacci Numbers Reconsidered 2(2)

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

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

where

```
take :: Integer -> [a] -> [a]
take 0 _            = []
take _ []           = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _
     = error "PreludeList.take: negative argument"
```

Summing up

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

```
fib :: Int -> Integer
fib n = last (take n fibs)
```

Even shorter:

```
fib :: Int -> Integer
fib n = fibs!!(n-1)
```

Remark:
Note the application of the

- Generator/Selector Principle

in this example.

Naive Evaluation (no sharing)

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

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

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

The Benefit of Lazy Evaluation (sharing) 1(3)

```
fibs ->> 0 : 1 : add fibs (tail fibs)
     ->> Introduc. abbrev. allows sharing of results
         0 : tf    (tf reminds to "tail of fibs")
         where tf = 1 : add 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
```

The Benefit of Lazy Evaluation (sharing) 2(3)

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

->> tf is only used at one place and can thus be
    eliminated
    0 : 1 : tf2
    where tf2 = 1 : tf3
                where tf3 = 2 : add tf2 tf3
```

The Benefit of Lazy Evaluation (sharing) 3(3)

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

Pitfall

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

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

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

This function again exposes

- exponential run-time and storage behaviour!

Crucial:

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

Illustration

```
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
```

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

Chapter 2.2

Stream Diagrams

Stream Diagrams

Problems on streams can often be considered and visualized as

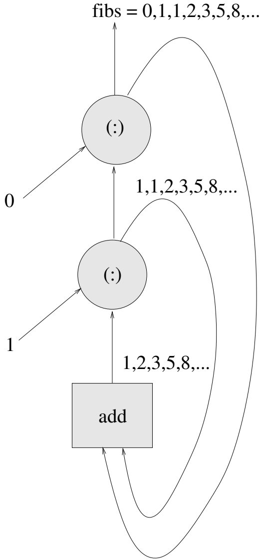
- processes.

In the following, we consider two examples:

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

Fibonacci Numbers

... as a stream diagram:



The Client/Server Application

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

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

```
reqs  =  client resps
resps =  server reqs
```

Implementation

```
type Request  = Integer
type Response = Integer
```

```
client ys = 1 : ys  (issues 1 as first request and
                     then each integer it receives
                     from the server)
server xs = map (+1) xs (adds 1 to each request it
                         receives)
```

The Client/Server Application (Cont'd)

Illustration: By stepwise evaluation

```
reqs ->> 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
```

The Client/Server Application (Cont'd)

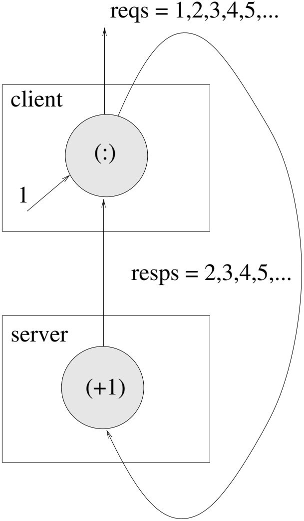
```
->> 1 : tr
    where tr = 2 : tr2
                   where tr2 = 3 : server tr2
->> 1 : 2 : tr2
    where tr2 = 3 : server tr2
->> ...
```

In particular, we obtain:

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

The Client/Server Application

... as a stream diagram:



Pitfall

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    (Obviously a trivial predicate)
```

The evaluation of:

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

...does not terminate!

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

Remedy: Lazy Patterns 1(3)

Ad-hoc Remedy:

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

- Replacing of pattern matching by an explicit usage of the selector function `head`.
- Moving the conditional inside of the list.

Remedy: Lazy Patterns 2(3)

Systematic remedy: Lazy patterns

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

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

Note: Even when using a lazy pattern the conditional must still be moved. But: The selector functions is avoided!

In practice, this can be very many selector functions that are saved this way making the programs "more" declarative and readable.

Remedy: Lazy Patterns 3(3)

Illustration: By stepwise evaluation

```
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

that benefits from

- stream programming.

Memoization

The concept of

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

Memo Functions, Memo Tables

A memo function is

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

A memo table allows

- to replace recomputation by table look-up.

Correctness of the overall approach:

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

Memo Functions, Memo Tables (Cont'd)

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

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

has to be defined such that the following equality holds:

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

A Concrete Approach with Memo Lists

Memo List:

The (generic) memo function/table

```
flist = [ f x | x <- [0..]]
```

...where f is a function on integers.

Application:
Each call of f is replaced by a look-up in flist.

Example 1: Computing Fibonacci Numbers

Computing Fibonacci numbers with memoization:

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

Compare this with the naive implementation of `fib`:

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

Note:

```
fibm n = fib n
```

Example 2: Computing Powers

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

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

Compare this with the naive implementation of `power`:

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

Observation:

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

  $\rightsquigarrow$ Significant performance gain!

Summing up

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

- is essentially the identity on functions but
- `memo` keeps track on the arguments, it has been applied to and the corresponding results
  Motto: look-up a result that has been computed previously instead of recomputing it!

Memo functions

- are not part of the Haskell standard, but there are nonstandard libraries

Summing up (Cont'd)

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:

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

Summing up (Cont'd)

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

- Chapter 19, Memoization
  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 (Cont'd)

- (Introduced streams without memoization)
  P. J. Landin. *A Correspondence between ALGOL60 and Church's Lambda-Notation: Part I*. Communications of the ACM, 8(2):89-101, 1965.
- (Extended Landin's streams with 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.

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.

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

- Stream programming
- Memoization

Avoiding Recomputations using Stream Prog.

- Computing Fibonacci numbers using stream prog.:

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

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

- Computing powers using stream programming:

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

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

- ...

Avoiding Recomputations using Memoization

- Computing Fibonacci numbers with memoization:

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

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

- Computing powers with memoization:

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

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

- ...

Summing up

Stream programming and memoization are

- no silver bullets

for improving performance by avoiding recomputations.

If, however, they hit they can

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

Obvious candidates

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

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

Silver Bullets exist Sometimes

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

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

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

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

Theorem

$$\forall\, n \in I\!N_0.\ fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Conclusion

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

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

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

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

Chapter 2: Further Reading (1)

- 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)

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

Chapter 2: Further Reading (6)

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

Motivation (Cont'd)

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

In detail:

- Top-down: starting from the initial problem, the algorithm works down to the solution by considering alternatives.
  - Divide-and-conquer
  - Backtracking search
  - Priority-first search
  - Greedy search
- Bottom-up: starting from small problem instances, the algorithm works up to the solution of the initial problem by combining solutions of smaller problem instances to solutions of larger ones.
  - Dynamic programming

Chapter 3.1

Divide-and-Conquer

Divide and Conquer

Given:

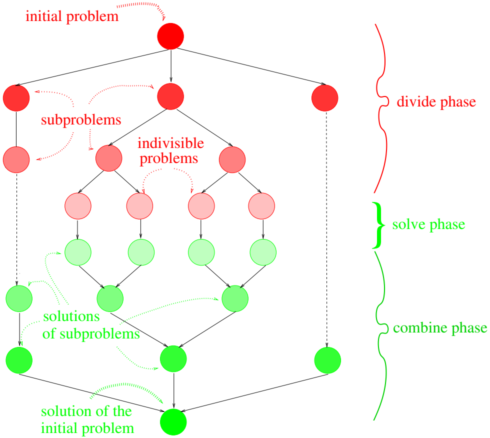
Let P be a problem specification.

Solving P – The Idea:

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

Illustrating the Divide-and-Conquer Principle

Successive stages in a divide-and-conquer algorithm:



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

Implementing Divide-and-Conquer as HOF (1)

The Initial Setting:

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

  and solutions with
  - solution instances of kind `s`

Objective:

- A higher-order function (HOF) `divideAndConquer`
  - solving suitably parameterized problem instances of kind `p` utilizing the "divide and conquer" principle.

Implementing Divide-and-Conquer as HOF (2)

The ingredients of `divideAndConquer`:

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

Implementing Divide-and-Conquer as HOF (3)

The HOF-Implementation:

```
divideAndConquer ::
 (p -> Bool) -> (p -> s) -> (p -> [p]) ->
                              (p -> [s] -> s) -> p -> s

divideAndConquer indiv solve divide combine initPb

 = dAC initPb
   where
    dAC pb
     | indiv pb  = solve pb
     | otherwise = combine pb (map dAC (divide pb))
```

Typical Applications of Divide-and-Conquer

Typical Applications:

- Application areas such as
  - Numerical analysis
  - cryptography
  - image processing
  - ...
- Quicksort
- Mergesort
- Binomial coefficients
- ...

Divide-and-Conquer for Quicksort

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

Pitfall

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

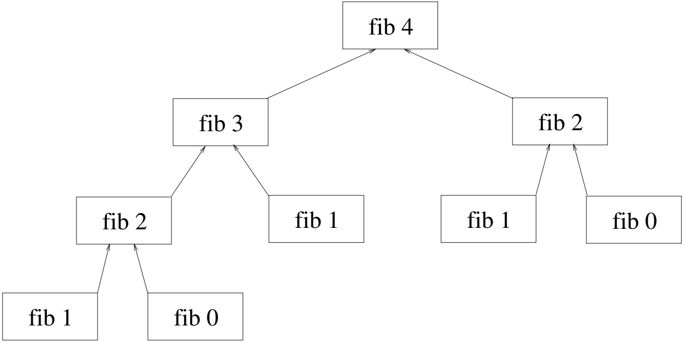
Consider:

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

...shows exponential runtime behaviour due to recomputations!

Illustration

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



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

Chapter 3.2

Backtracking Search

Backtracking Search

Given:

Let P be a problem specification.

Solving P – The Idea

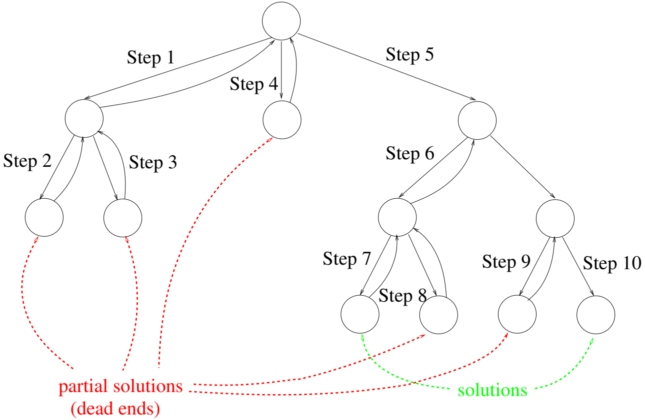
- Search for a particular solution of the problem by a systematic trial-and-error exploration of the solution space.

Main Problem Characteristics for Applicability

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

Illustrating Backtracking Search

General stages in a backtracking algorithm:



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

Illustrating Backtracking Search (Cont'd)

Intuitively

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

Note

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

Implementing Backtracking Search as HOF (1)

The Initial Setting:

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

  and solutions with
  - solution instances of kind `s`

Objective:

- A higher-order function (HOF) `searchDfs`
  - solving suitably parameterized problem instances of kind `p` utilizing the "backtracking" principle.

Implementing Backtracking Search as HOF (2)

Often:

- The search space is large.

Hence, the graph representing the search space

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

This reqires:

- An appropriate type `node` that represents node information
- a successor function `succ` of type `node -> [node]` that generates the list of successors of a node.

Implementing Backtracking Search as HOF (2)

Assumptions:

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

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

The ingredients of `searchDfs`:

- `node`: A type representing node information.
- `succ :: node -> [nodes]`: The function `succ` yields the list of successors of a node.
- `goal :: node -> Bool`: The function `goal` determines if a node is a solution.

Implementing Backtracking Search as HOF (3)

The HOF-Implementation:

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

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

The Abstract Data Type Stack (1)

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

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

push       :: a -> Stack a -> Stack a
pop        :: Stack a -> Stack a
top        :: Stack a -> a
emptyStack :: Stack a
stackEmpty :: Stack a -> Bool
```

The Abstract Data Type Stack (2)

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

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

push x s = Stk x s

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

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

emptyStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _        = False
```

The Abstract Data Type Stack (3)

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

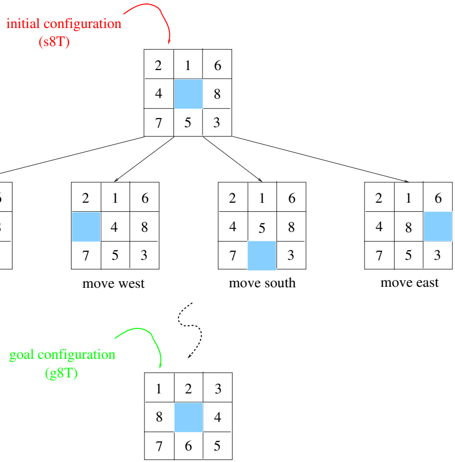
```
newtype Stack a = Stk [a]

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

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

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

emptyStack = Stk []

stackEmpty (Stk []) = True
stackEmpty (Stk _)  = False
```

Typical Applications of Backtrackting Search

Typical Applications:

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

The Eight-Tile Problem



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

A Backtracking Search 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 for 8TP (2)

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

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

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

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

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

A Backtracking Search for 8TP (3)

Modeling nodes in the search graph:

```
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 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) that lead to the goal configuration:

```
dfs8Tile :: [[Position]]
dfs8Tile = map elems ls
 where ((BDS ls):_)
    = searchDfs suc8Tile goal8Tile (BDS [s8T])
```

Chapter 3.3

Priority-first Search

Priority-first Search

Given:

Let P be a problem specification.

Solving P – The Idea

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

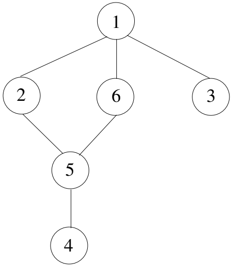
Note: In contrast to plain backtracking search, which proceeds unguided and can thus be considered blind, priority-first search/best-first search benefits from (hopefully correct) information pointing it towards "more promising" nodes.

Priority-first Search (Cont'd)

Main Problem Characteristics for Applicability

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

Illustrating Search Strategies



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

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

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

Implementing Priority-first Search as HOF (1)

The Initial Setting:

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

  and solutions with
  - solution instances of kind `s`

Objective:

- A higher-order function (HOF) `searchPfs`
  - solving suitably parameterized problem instances of kind `p` utilizing the "priority-first/best-first" principle.

Implementing Priority-first Search as HOF (2)

Assumptions:

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

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

The ingredients of `searchPfs`:

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

Implementing Priority-first Search as HOF (3)

The HOF-Implementation:

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

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

The Abstract Data Type PQueue (1)

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

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

emptyPQ :: PQueue a
pqEmpty :: PQueue a -> Bool
enPQ    :: (Ord a) => a -> PQueue a -> PQueue a
dePQ    :: (Ord a) => PQueue a -> PQueue a
frontPQ :: (Ord a) => PQueue a -> a
```

The Abstract Data Type PQueue (2)

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

```
newtype PQueue a = PQ [a]

emptyPQ = PQ []

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

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

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

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

Typical Applications of Priority-first Search

Typical Applications:

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

A Priority-first Search for 8TP

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

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

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

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

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

Chapter 3.4

Greedy Search

Greedy Search

Given:

Let P be a problem specification.

Solving P – The Idea

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

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

Greedy Search (Cont'd)

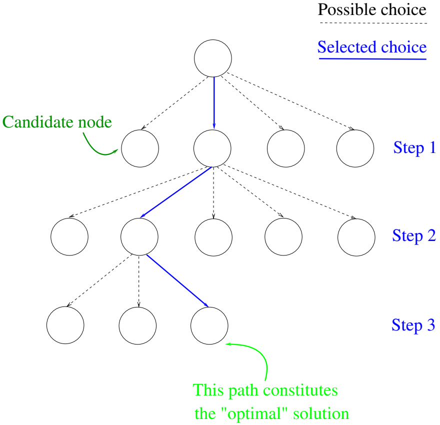
Main Problem Characteristics for Applicability

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

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

Illustrating Greedy Search

Successive stages in a greedy algorithm:



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

Implementing Greedy Search as HOF (1)

The Initial Setting:

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

  and solutions with
  - solution instances of kind `s`

Objective:

- A higher-order function (HOF) `searchGreedy`
  - solving suitably parameterized problem instances of kind `p` utilizing the "greedy/hill climbing" principle.

Implementing Greedy Search as HOF (2)

Assumptions:

- an acyclic implicit graph
- no local minimums, i.e., no locally best solutions

The ingredients of `searchGreedy`:

- `node`: A type representing node information.
- `<=`: A comparison criterion for nodes; usually, this is the relator `<=` of the type class `Ord`.
- `succ :: node -> [nodes]`: The function `succ` yields the list of successors of a node.
- `goal :: node -> Bool`: The function `goal` determines if a node is a solution.

Implementing Greedy Search as HOF (3)

The HOF-Implementation:

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

searchGreedy succ goal x
 = search' (enPQ x emptyPQ)
   where
    search' q
     | pqEmpty q         = []
     | goal (frontPQ q) = [frontPQ q]
     | otherwise
         = let x = frontPQ q
           in search' (foldr enPQ emptyPQ (succ x))
```

Implementing Greedy Search as HOF (4)

Note:

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

Typical Applications of Greedy Search

Typical Applications:

- Graph algorithms, e.g., Prim's minimum spanning tree algorithm
- Money Change Problem (MCP)
- ...

A Greedy Search for MCP

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

Modeling coins:

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

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

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

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

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

A Greedy Search for MCP (Cont'd)

The goal function:

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

Putting things together:

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

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

Note: For `coins = [1,3,6,12,2,30]` the above algorithm can yield suboptimal solutions: E.g., `change 48 ->> [30,12,6]` instead of the optimal solution `[24,24]`.

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

Dynamic Programming

Dynamic Programming

Given:

Let P be a problem specification.

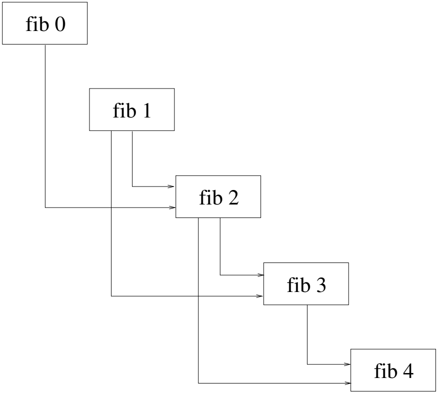
Solving P – The Idea

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

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

Illustrating Dynamic Programming for fib

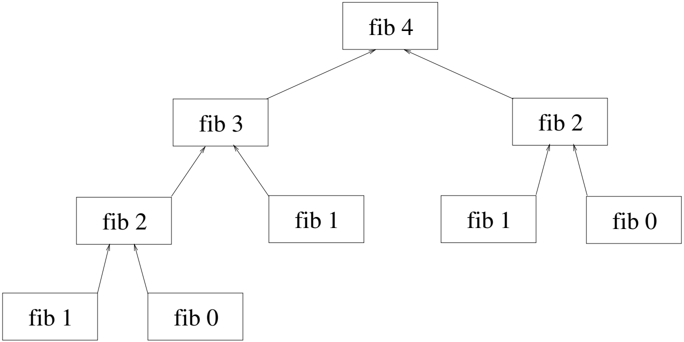
The dynamic programming computation of the Fibonacci numbers (no recomputation of the solution to subproblems!):



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 (recomputing the solution to many subproblems!):



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

Implementing Dynamic Programming as HOF (1)

The Initial Setting:

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

  and solutions with
  - solution instances of kind `s`

Objective:

- A higher-order function (HOF) `dynamic`
  - solving suitably parameterized problem instances of kind `p` utilizing the "dynamic programming" principle.

Implementing Dynamic Programming as HOF (2)

The ingredients of the HOF `dynamic`:

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

Implementing Dynamic Programming as HOF (3)

The HOF-Implementation:

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

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

The Abstract Data Type Table (1)

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

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

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

Note:

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

The Abstract Data Type Table (2)

A user-invisible implementation of Table as an Array:

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

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

findTable (Tbl a) i = a ! i

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

The Abstract Data Type `Table` (2)

Note:

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

Typical Applications of Dynamic Programming

Typical Applications:

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

Computing Fibonacci Numbers using Dynamic Programming

Defining the problem-dependent parameters:

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

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

Putting things together:

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

Comparing Dynamic Programming and Memoization

Overall

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

Conceptual difference

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

Comparing Dynamic Programming and Memoization (Cont'd)

Minor benefits of dynamic programming

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

Comparing Dynamic Programming and Memoization (Cont'd)

Minor benefits of memoization

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

Chapter 3.5: Further Reading (1)

- 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 (2)

- 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 (3)

- 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 (4)

- 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)

Functional Programming

- The usage of `=` in functional definitions of the type

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

  as e.g. used in Haskell in the definition of a function `f` are genuine mathematical equations.
- The equations state that the expressions on the left hand side and the right hand side have the same value.

Functional vs. Imperative Programming (2)

Imperative Programming

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

```
x = x+y
```

  does not mean that `x` and `x+y` have the same value.
- Here, = is used to denote a command, a destructive assignment statement meaning that the old value of `x` is destroyed and replaced by the value of `x+y`.

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

Consequence

Reasoning about

- functional definitions

is because of this difference a lot easier as about

- programs using destructive assignments

For functional definitions

- standard (algebraic) reasoning about mathematical equations applies.

For example: The sequence of definitions in Haskell

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

raises an error `"x" multiply defined` since `=` in Haskell has the meaning "is by definition equal to"; redefinition is forbidden.

Illustrating Algebraic Reasoning

By algebraic reasoning on equations we obtain:

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

Proof:

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

Extending Algebraic Reasoning to Functional Definitions

First Example:

This allows us to conclude: 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.

Reasoning on Functional Definitions – More Examples (1)

Second Example:

Let

```
a = 3
b = 4

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

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

`f a (f a b)` has value 634.

Reasoning on Functional Definitions – More Examples (2)

Proof:

$$\begin{aligned} f\ a\ (f\ a\ b) &= f\ a\ (a^2 + b^2) \\ &= f\ 3\ (3^2 + 4^2) \\ &= f\ 3\ (9 + 16) \\ &= f\ 3\ 25 \\ &= 3^2 + 25^2 \\ &= 9 + 625 \\ &= 634 \end{aligned}$$

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

Reasoning on Functional Definitions – More Examples (3)

Third Example:

Let

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

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

By equational reasoning on the functional definitions of `g` and `h` we can show the equality of the Haskell expressions

`h (a+b) (a-b)` and `g a b`.

Reasoning on Functional Definitions – More Examples (4)

Proof:

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

Remark (1)

We have:

In equational reasoning functions can be applied/unapplied

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

Remark (2)

Note: Some care needs to be taken though. Let

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

The first equation `isZero 0 = True`

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

The second equation, however, `isZero n = False` can not, since Haskell implicitly imposes an ordering on the equations:

- Application from left-to-right (i.e., replacing `isZero n` by `False`), and from right-to-left (i.e., replacing `False` by `isZero n` for some `n`) is legal only, if `n` is different from `0`.

Reasoning on Functional Definitions – More Examples (5)

Fourth Example:

The standard implementation of the `reverse` function

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

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

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

Reasoning on Functional Definitions – More Examples (6)

A more efficient implementation of the functionality of the `reverse` function is

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

Reasoning on Functional Definitions – More Examples (7)

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

The Haskell expressions

`reverse xs` and `fastReverse xs`

are equal for all finite lists `xs`.

Summing up

Functional definitions are

- genuine mathematical equations.

This allow us to prove

- equality and other relations of functional expressions

by applying standard algebraic mathematical reasoning.

In particular, this can be used to replace

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

  Examples:

  - Basic: Replace `(x*y)+(x*z)` by `x*(y+z)`
  - Advanced: Replace `reverse` by `fastReverse`

Chapter 4.2

Functional Pearls

Functional Pearls – The Very Idea (1)

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

- evolves from calculation!

In more detail:

Starting from a problem with a

- simple, intuitive but often inefficient specification

we shall arrive at an

- efficient though often more complex and possibly less intuitive implementation

by means of

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

  Example: From `reverse` to `fastReverse`.

Functional Pearls – The Very Idea (2)

It is important to note:

The functional pearl

- is not the final (efficient) implementation
- but the calculation process leading to it!

Functional Pearls – Origin and Background (1)

In the course of founding the

- *Journal of Functional Programming*

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

- Functional Pearls

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

- Programming Pearls

in the

- *Communications of the ACM*

Functional Pearls – Origin and Background (2)

Since 1990, some

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

  and other problems.

Some more appeared in proceedings of conferences including editions of the

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

Functional Pearls – Origin and Background (3)

Roughly,

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

In his recent monograph

- *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.

Outline

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

- The Smallest Free Number
- Not the Maximum Segment Sum
- A Simple Sudoku Solver

Last but not least

It is worth noting:

The name of the functional programming language

- GoFER

is an acronym for

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

Chapter 4.3

The Smallest Free Number

The Smallest Free Number (SFN) Problem

The SFN-Problem:

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

Examples:

The smallest free number for

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

Analyzing the Problem

Obviously

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

Example:

Computing the smallest free number for the set X

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

Simple Algorithm for solving the SFN-Problem

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

The simple SFNP-Algorithm:

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

Possible Implementation of the Simple Algorithm

...by means of a system of functions

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

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

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

The Advanced Algorithmic Problem

The simple SFNP-Algorithm is sound but inefficient:

- Sorting is not of linear time complexity.

The Advanced SFNP-Algorithm Problem:

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

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

Towards the Linear Time Algorithm

The SFN-problem can be specified as a function `minfree`, defined by

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

with

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

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

```
type Nat = Int
```

the type of natural numbers starting from 0.

Analysing `minfree`

The function `minfree` solves the SFN-problem but its evaluation requires on a list of length n

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

For illustration consider:

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 `minfree` we will develop an

- array based and a
- divide-and-conquer based

linear time algorithm for the SFN-problem.

The key fact (KF) both algorithms rely on is:

- There is a number in `[0..length xs]` that is not in `xs`

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

This implies:

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

Towards the Array-Based Algorithm

The array-based algorithm uses KF to build a

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

The checklist is a

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

Algorithmic idea:

- For each element `x` in `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 Algorithm

The array-based 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: This algorithm

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

Two Variants of the Array-Based Algorithm (1)

1st Variant: The function accumArray can be used to

- sort a list of numbers in linear time, provided the elements of the list all lie in some known range.

This allows

- replacing of checklist by countlist.

```
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.

Two Variants of the Array-Based Algorithm (2)

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

- using a constant-time array update operation.

In Haskell, this can be done using a suitable monad, such as the

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

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

Note, however: This variant is essentially

- a procedural program in functional clothing.

Towards the Divide-and-Conquer Algorithm (1)

Algorithmic idea:

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

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

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

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

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

Towards the Divide-and-Conquer Algorithm (2)

Going on, choose any natural number `b`, and let

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

then

- `as` and `vs` are disjoint, and `bs` and `us` are disjoint.

This implies:

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

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

The Divide-and-Conquer Algorithm

Moreover, because of

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

we obtain (still for any natural number b):

The Basic Divide-and-Conquer Algorithm:

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

Refining the Divide-and-Conquer Algorithm (1)

Note, the straightforward evaluation of the test

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

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

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

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

```
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.

Refining the Divide-and-Conquer Algorithm (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.

Refining the Divide-and-Conquer Algorithm (3)

Provided `b` is chosen so that both

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

the below recursive definition of `minfree` is well-founded:

```
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
```

Refining the Divide-and-Conquer Algorithm (4)

It remains to choose `b`.

This choice shall ensure:

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

This is achieved by choosing `b` as

```
b = a + 1 + n `div` 2
```

where `n = length xs`.

Refining the Divide-and-Conquer Algorithm (5)

If `n /= 0` and `length us < b-a`, then

- `(length us) <= (n div 2) < n`

And, if `length us = b-a`, then

- `(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 Divide-and-Conquer Algorithm

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

The Optimized Divide-and-Conquer Algorithm:

```
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
```

Summing up

The optimized divide-and-conquer algorithm is about

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

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

- It can be considered a simplification of the common programming task to find some object not in use: Numbers then name objects, and X the set of objects that are currently in use.

Summing up (Cont'd)

For a "procedural" programmer

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

For a "pure functional" programmer

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

This explains

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

Sometimes, however, this gap

- vanishes as for the SFN-problem.

Chapter 4.4

Not the Maximum Segment Sum

Background and Motivation

A segment of a list

- is a contiguous subsequence.

The Maximum Segment Sum (MSS) Problem:

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

Example:

Let L be the list

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

The maximum segment sum of L is

- 3 (from the segment `[2,1]`).

Background and Motivation (Cont'd)

The MSS-problem

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

In this pearl, however,

- we consider the "Maximum Non-Segment Sum (MNSS) Problem".

The Maximum Non-Segment Sum (MNSS) Problem

A non-segment of a list

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

The Maximum Non-Segment Sum (MNSS) Problem:

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

Example:

Let L be the list

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

The maximum non-segment sum of L is

- 2 (from the non-segment `[2,1,-1]`).

What does MNSS make a Pearl Problem?

It is worth noting:

Let L be a list of length n.

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

Hence

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

This raises the problem

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

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

Specifying Solution of the MNSS-Problem

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

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

Intuition:

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

The Implementation of nonsegs

The implementation of the function nonsegs

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

relies on the supporting functions

- extract
- markings

which itself relies on the supporting function

- booleans

The Implementation of nonsegs (Cont'd)

The implementation of the supporting functions:

```
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)
```

The Implementation of nonsegs (Cont'd)

Intuition underlying the supporting functions:

To define the function `nonsegs`

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

This marking

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

Then

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

The Implementation of `nonsegs` (Cont'd)

The function

- `nonseg :: [(a,Bool)] -> Bool`, finally, returns `True` on a list `xms` iff `map snd xsm` describes a non-segment marking (its implementation 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
```

Intuition:

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

- State `E` (for Empty), starting state: if in `E`, markings only in the set F^* have been recognized.
- State `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.
- State `M` (for Middle): if in state `M`, this indicates the processing of markings in the set $F^* T^+ F^+$, i.e., a middle segment.
- State `N` (for Non-segment): if in state `N`, this indicates the processing of non-segments markings.

The Finite State Automaton (Cont'd)

This allows us to define:

```
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
```

It is worth noting:

- 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.

Towards Deriving the Linear Time Algorithm

Recall first the specifying (initial) 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.

Deriving 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
```

We have:

- `nonsegs == pick N`

Properties of pick

Moreover, we can prove

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

the equalities:

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

Deriving the Linear Time Algorithm (2)

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

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

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

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

```
pickall = 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.

By means of 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`

The 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 the Application of the Fusion Law (1)

...in our scenario 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 -∞, we have:

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

...which gives the definition of b.

Towards the Application of 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 three components is similar.

Towards the Application of the Fusion Law (3)

max is used as an abbreviation for maximum:

$$
\begin{array}{ll}
 & max\ (map\ sum\ (nss\ dpl\ map\ (\mathbin{+\!\!+}\ [x])\ (nss \mathbin{+\!\!+} mss))) \\
= & \text{(definition of } map\text{)} \\
 & max\ (map\ sum\ nss \mathbin{+\!\!+}\ map\ (sum\ .\ (\mathbin{+\!\!+}[x]))(nss \mathbin{+\!\!+} mss)) \\
= & \text{(since } sum\ .\ (\mathbin{+\!\!+}[x]) = (+x)\ .\ sum\text{)} \\
 & max\ (map\ sum\ nss \mathbin{+\!\!+} map\ ((+x)\ .\ sum)\ nss \mathbin{+\!\!+} mss)) \\
= & \text{(since } max\ (xs \mathbin{+\!\!+} ys) = (max\ xs)\ \text{max}\ (max\ ys)\text{)} \\
 & max\ (map\ sum\ nss)\ \text{max}\ max\ (map\ ((+x)\ .\ sum)\ (nss \mathbin{+\!\!+} mss)) \\
= & \text{(since } max\ .\ map\ (+x) = (+x)\ .\ max\text{)} \\
 & max\ (map\ sum\ nss)\ \text{max}\ (max\ (map\ sum\ (nss \mathbin{+\!\!+} mss)) + x) \\
= & \text{(introducing } n = max\ (map\ sum\ nss) \text{ and} \\
 & \qquad\qquad m = max\ (map\ sum\ mss)\text{)} \\
 & n\ max\ ((n\ max\ m) + x)
\end{array}
$$

Towards the Application of the Fusion Law (4)

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

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

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

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

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)
```

Concluding Remarks (1)

The MSS problem goes back to Bentley:

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

Gries and 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.

Concluding Remarks (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

Towards the Specifying Solution (1)

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')
```

Towards the Specifying Solution (2)

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

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

Towards the Specifying Solution (3)

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

1. 1st Approach:
   - Construct a list of all correctly completed grids.
   - Then test the input grid against them to identify those whose non-blank entries match the given ones.
2. 2nd Approach:
   - Start with the input grid and construct all possible choices for the blank entries.
   - Then compute all grids that arise from making every possible choice and filter the result for the valid ones.

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

Specifying Solution of the Sudoku-Problem (1)

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

```
solve = filter valid . expand . choices

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

Intuition:

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

Specifying Solution of the Sudoku-Problem (2)

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

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

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

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

Specifying Solution of the Sudoku-Problem (3)

The implementation of `choices`:

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

Intuition:

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

Specifying Solution of the Sudoku-Problem (4)

The implementation of expand:

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

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

Intuition:

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

Specifying Solution of the Sudoku-Problem (5)

The implementation of `valid`:

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

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

Intuition:

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

Specifying Solution of the Sudoku-Problem (6)

The implementation of rows and columns:

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

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

Intuition:

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

Specifying Solution of the Sudoku-Problem (7)

The implementation of 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
```

Intuition:

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

Specifying Solution of the Sudoku-Problem (8)

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

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

Wholemeal Programming

Instead of

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

the present approach has gone for functions that

- treat the matrix as a complete entity in itself.

Geraint Jones coined the notion

- wholemeal programming

for this style of programming.

Wholemeal programming helps

- avoiding indexitis and
- encourages lawful program construction.

Lawful Programming

Example:

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

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

  This means, all 3 functions are involutions.

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

  ```
  map rows . expand = expand . rows          (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

to be constructed and checked for validity!

This is hopeless!

Towards a Better Performing Algorithm

Pruning the matrix of choices:

Idea

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

Hence, we are seeking for a function

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

that satisfies

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

and realizes the above idea.

Towards defining prune

Pruning a row

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

where

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

Intuition:

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

Laws for pruneRow, nodeups, and cp

- The function pruneRow satisfies law (G):

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

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

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

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

Rewriting `filter valid . expand`

We can prove:

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

It is worth noting:

- The order of the 3 filters on the right hand side above is not relevant.

Work plan:

- Apply each of the filters to `expand`.

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

Reasoning in the boxs Case (1)

$$
\begin{aligned}
& \mathit{filter}\ (\mathit{all}\ \mathit{nodups}\ .\ \mathit{boxs})\ .\ \mathit{expand} \\
= \ & \{\text{(H), since } \mathit{boxs}\ .\ \mathit{boxs} = \mathit{id}\} \\
& \mathit{map}\ \mathit{boxs}\ .\ \mathit{filter}\ (\mathit{all}\ \mathit{nodups})\ .\ \mathit{map}\ \mathit{boxs}\ .\ \mathit{expand} \\
= \ & \{\text{(F)}\} \\
& \mathit{map}\ \mathit{boxs}\ .\ \mathit{filter}\ (\mathit{all}\ \mathit{nodups})\ .\ \mathit{expand}\ \mathit{boxs} \\
= \ & \{\text{definition of } \mathit{expand}\} \\
& \mathit{map}\ \mathit{boxs}\ .\ \mathit{filter}\ (\mathit{all}\ \mathit{nodups})\ .\ \mathit{cp}\ .\ \mathit{map}\ \mathit{cp}\ .\ \mathit{boxs} \\
= \ & \{\text{(I), and } \mathit{map}\ f\ .\ \mathit{map}\ g = \mathit{map}\ (f\ .\ g)\} \\
& \mathit{map}\ \mathit{boxs}\ .\ \mathit{cp}\ .\ \mathit{map}\ (\mathit{filter}\ \mathit{nodups}\ .\ \mathit{cp})\ .\ \mathit{boxs} \\
= \ & \{\text{(G)}\} \\
& \mathit{map}\ \mathit{boxs}\ .\ \mathit{cp}\ .\ \mathit{map}\ (\mathit{filter}\ \mathit{nodups}\ .\ \mathit{cp}\ .\ \mathit{pruneRow})\ .\ \mathit{boxs}
\end{aligned}
$$

Reasoning in the boxs Case (2)

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

Summing up

- We have shown:

```
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:

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

  where

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

2nd and Improved Implementation of `solve`

The Pruning-improved Implementation of `solve`:

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

It is worth noting:

Pruning can be done more than once.

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

Tuning the Solver Further

Idea

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

  ⇝ single-cell expansion

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

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

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

Towards defining expand1

Which cell to expand?

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

Note:

- If there is a cell with no choices then the Sudoku problem is unsolvable.

  (From a pragmatic point of view, such cells should be identified quickly.)

Defining expand1

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

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

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

Remarks on `expand1`

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

Completeness and Safety of a Matrix

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

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

This suggests:

A matrix is

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

It is worth noting:

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

Completeness and Safety Tests

Completeness and safety can be tested as follows.

- Completeness Test:

  ```
  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)
  ```

  where

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

We can show

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

$$
\begin{aligned}
& \mathit{filter}\ \mathit{valid}\ .\ \mathit{expand} \\
= \ & \{\text{since } \mathit{expand} = \mathit{concat}\ .\ \mathit{map}\ \mathit{expand}\ .\ \mathit{expand}1 \\
& \text{on incomplete matrices}\} \\
& \mathit{filter}\ \mathit{valid}\ .\ \mathit{concat}\ .\ \mathit{map}\ \mathit{expand}\ .\ \mathit{expand}1 \\
= \ & \{\text{since } \mathit{filter}\ p\ .\ \mathit{concat} = \mathit{concat}\ .\ \mathit{map}\ (\mathit{filter}\ p)\} \\
& \mathit{concat}\ .\ \mathit{map}\ (\mathit{filter}\ \mathit{valid}\ .\ \mathit{expand})\ .\ \mathit{expand}1 \\
= \ & \{\text{since } \mathit{filter}\ \mathit{valid}\ .\ \mathit{expand} = \mathit{filter}\ \mathit{valid}\ .\ \mathit{expand}\ .\ \mathit{prune}\} \\
& \mathit{concat}\ .\ \mathit{map}\ (\mathit{filter}\ \mathit{valid}\ .\ \mathit{expand}\ .\ \mathit{prune})\ .\ \mathit{expand}1
\end{aligned}
$$

3rd and Final Implementation of `solve`

Introducing

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

we have on safe but incomplete matrices that

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

This allows:

The Final Implementation of `solve`:

```
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: 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, 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. *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.
- Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. Journal of Functional Programming 19(1):95-106, 2009.
- Colin Runciman. *Lazy Wheel Sieves and Spirals of Primes*. Journal of Functional Programming 7(2):219-225, 1997.

Part III

Quality Assurance

Chapter 5

Testing

Objective

How can we gain (sufficiently much) confidence that

- ours and
- other people's programs

are sound?

Essentially, there are two means at our disposal:

- Verification
- Testing

Verification vs. Testing

- Verification
  - Formal soundness proof (soundness of the specification, soundness of the implementation).
  - High confidence but often high effort.
- Testing
  - Two Variants
    - Ad hoc: Controllable effort but usually unquantifiable, questionable quality statement.
    - Systematically: Controllable effort with quantifiable quality statement.

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

Edsger Dijkstra

On the other hand, testing is often

- amazingly successful in revealing errors.

Minimum Requirements of Testing

(Systematic) testing of programs should be

- Specification-based
- Tool-supported
- Automatically

Minimum Requirements of Testing (Cont'd)

There shall be reporting on

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

Desirable, too

- Reproducibility of tests
- Repeated testing after program modifications

Program Specification

Inevitable

- Specification of the meaning of the program
  - Informally (e.g., as commentary in the program, in a separate documentation)
    $\leadsto$ disadvantage: often ambiguous, open to interpretation
  - Formally (e.g., in terms of pre- and post-conditions, in a formal specification language)
    $\leadsto$ advantage: precise and rigorous, unambiguous

In this chapter

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

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

It is worth noting

QuickCheck and its specification and test data generator languages are:

- Examples of so-called domain-specific embedded languages
  $\rightsquigarrow$ special strength of functional programming.
- Implemented as a combinator library in Haskell
  $\rightsquigarrow$ allows us to make use of the full expressiveness of Haskell when defining properties and test data generators.
- Part of the standard Haskell-distribution (for both GHC and Hugs; see module `QuickCheck`)
  $\rightsquigarrow$ ensures easy and direct usability.

Chapter 5.1

Property Definitions

Simple Property Definition w/ QuickCheck (1)

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

Example:

Define inside of the program the property

```
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
```

Simple Property Definition w/ QuickCheck (2)

Note:

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

Simple Property Definition w/ QuickCheck (3)

The same example slightly varied:

Define inside of the program the property

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

Double-checking the property with Hugs yields:

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

Simple Property Definition w/ QuickCheck (4)

Note:

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

The error report contains:

- The number of tests successfully passed
- A counter example

Advanced Property Definition (1)

Given:

- A function `insert`
- A predicate `ordered`

Property under test:

- Insertion into a sorted list

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

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

However, this property is falsifiable.

- The definition is naive and too strong (note that `xs` is not supposed to be sorted).

Advanced Property Definition (2)

First fix (trial-and-error):

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

Note:

- `ordered xs ==>`: This adds a precondition to the property definition.

  ⇝ Generated test data that do not match the precondition, are dropped.
- `==>`: is not a simple Boolean operator but affects the selection of test data.

  ⇝ Property definitions that rely on such operators always have the result type `Property` in QuickCheck.
- Overall: A trial-and-error approach to generating test data: Generate, then check if usable; if not, drop.

Advanced Property Definition (3)

Second Fix (systematic):

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

Note:

- This fix works by direct quantifying (in the running example: direct quantifying over sorted lists)
- Overall: A systematic approach to generating test data: Only useful test data are generated.

The Operator ($) — A Quick Reminder

Standard Prelude:

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

Remark:

- The operator `($)` is Haskell's infix function application.
- It is useful to avoid the usage of parentheses:

  Example: `f (g x)` can be written as `f $ g x`.

It is worth noting

Expressiveness:

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

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

Testing multiple properties:

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

- `>quickCheck Module.hs`

in order to test all properties defined in `Module.hs` at once.

Chapter 5.2

Testing against Abstract Models

Idea

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

- abstract model (reference model)

In the following:

- Demonstrating this by an extended example: Developing an abstract data type for queues.

Abstract Model of Queues

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

Specification:

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

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

The Concrete Model of Queues (1)

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

Basic Idea:

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

Together this ensures:

- Efficient access to list front and list back
  ⇝ ++ for addition boils down to : (strength reduction)

Example:

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

The Concrete Model of Queues (2)

Implementation:

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

In the following

We think of

- `Queue` and
- `QueueI`

in terms of

- specification and
- implementation

of FIFO queues, respectively.

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

Relating Queues and Abstract Queues

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

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

The function `retrieve`

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

Soundness Properties for Operations on QueueI

The understanding of `QueueI` and `Queue` as lists on integers

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

By means of `retrieve` we can double-check, if

- the results of applying the efficient operations on `QueueI` coincide with those of the abstract operations on `Queue`.

Soundness Properties: Initial Definitions (1)

The below properties can reasonably be expected to hold:

```
prop_empty     = retrieve emptyI == empty
prop_add x q   = retrieve (addI x q)
                          == add x (retrieve q)
prop_isEmpty q = isEmptyI q == isEmpty (retrieve q)
prop_front q   = frontI q   == front (retrieve q)
prop_remove q  = retrieve (removeI q)
                          == remove (retrieve q)
```

However, this is not true!

Soundness Properties: Initial Definitions (2)

Testing e.g. prop_isEmpty using QuickCheck yields:

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

Problem:

- The specification of isEmpty assumes implicitly that the following invariant holds:
  - The front of the list is only empty, if the back of the list is empty, too:

    ```
    isEmptyI (f,b) = null f
    ```

Soundness Properties: Initial Definitions (3)

In fact:

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

This silent assumption has to be made explicit in terms of an invariant.

Soundness Properties: Refined Definitions (1)

We define the invariant as follows:

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

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

```
prop_empty     = retrieve emptyI == empty
prop_add x q   = invariant q ==>
        retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = invariant q ==>
        isEmptyI q == isEmpty (retrieve q)
prop_front q   = invariant q ==>
        frontI q == front (retrieve q)
prop_remove q  = invariant q ==>
        retrieve (removeI q) == remove (retrieve q)
```

Soundness Properties: Refined Definitions (2)

Now, testing prop_isEmpty using QuickCheck yields:

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

However, testing prop_front still fails:

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

Problem:

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

Soundness Properties: Final Definitions

Fix:

- Add `not (isEmptyI q)` to the preconditions of the relevant properties.

This leads to:

```
prop_empty      = retrieve emptyI == empty
prop_add x q    = invariant q ==>
        retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = invariant q ==>
        isEmptyI q == isEmpty (retrieve q)
prop_front q    = invariant q && not (isEmptyI q) ==>
        frontI q == front (retrieve q)
prop_remove q  = invariant q && not (isEmptyI q) ==>
        retrieve (removeI q) == remove (retrieve q)
```

Now:

- All properties pass the test successfully!

Soundness Considerations Continued

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

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

Note:

So far we only tested that

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

```
invariant (f,b) = not (null f) || null b
```

Adding Missing Soundness Properties (1)

Defining properties for operations producing queues:

```
prop_inv_empty    = invariant emptyI
prop_inv_add x q  = invariant q ==>
                      invariant (addI x q)
prop_inv_remove q = invariant q &&
                    not (isEmptyI q) ==>
                      invariant (removeI q)
```

Adding Missing Soundness Properties (2)

Testing by means of QuickCheck yields:

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

Problem:

- The invariant must hold
  - not only after applying removeI,
  - but also after applying addI to the empty list; adding to the back of a queue breaks the invariant in this case.

Soundness Properties: Completed Now!

To overcome the last and final problem:

- Adjust the function addI as follows:

```
addI x (f,b) = flipQ (f,x:b)
-- instead of: addI x (f,b) = (f,x:b)
```

  with flipQ as defined previously.

Now:

- All properties pass the test successfully!

Summing up

In the course of developing this example it turned out:

- Testing revealed (only) one bug in the implementation (this was in function `addI`).
- But: Several missing preconditions and a missing invariant in the original definitions of properties were found and added.

Both is typical and valuable:

- The additional conditions and invariants are now explicitly given in the program text.
- They add to understanding the program and are valuable as documentation, both for the program developer and for future users (think e.g. of program maintainance!).

Chapter 5.3

Testing against Algebraic Specifications

Algebraic Specifications

Testing against algebraic specifications is (often) a useful alternative to testing against an abstract model.

An algebraic specification

- provides equational constraints the operations ought to satisfy.

Algebraic Specifications

For FIFO queues, e.g., we might start with the following algebraic specifications:

```
prop_isEmpty q       = invariant q ==>
          isEmptyI q == (q == emptyI)
prop_front_empty x  = frontI (addI x emptyI) == x
prop_front_add x q  = invariant q &&
                      not (isEmptyI q) ==>
          frontI (addI x q) == frontI q
prop_remove_empty x =
          removeI (addI x emptyI) == emptyI
prop_remove_add x q = invariant q &&
                      not (isEmptyI q) ==>
          removeI (addI x q) == addI x (removeI q)
```

Testing Algebraic Specifications (1)

Testing `prop_remove_add` using `QuickCheck` yields:

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

Problem:

- The left hand side, i.e., `removeI (addI x q)`, yields: `([0,0],[])`
- The right hand side, i.e., `addI x (removeI q)`, yields: `([0],[0])`
- The queue representations `([0,0],[])` and `([0],[0])` are equivalent (representing both the abstract queue `[0,0]`) but are not equal!

Testing Algebraic Specifications (2)

Fix:

- Consider "equivalent" instead of "equal":

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

In fact: Replacing

```
prop_remove_add x q = invariant q &&
                      not (isEmptyI q) ==>
      removeI (addI x q) == addI x (removeI q)
```

by

```
prop_remove_add x q = invariant q &&
                      not (isEmptyI q) ==>
      removeI (addI x q) 'equiv' addI x (removeI q)
```

yields as desired:

- The test of prop_remove_add passes successfully!

Testing Algebraic Specifications (3)

Similar to the setup in Chapter 5.1, we have to check:

- All operations producing queues yield results that are equivalent, if the arguments are.

Example:

For the operation `addI` this can be expressed by:

```
prop_add_equiv q q' x = q 'equiv' q' ==>
                   addI x q 'equiv' addI x q'
```

Summing up

Though mathematically sound, the definition of `prop_add_equiv` is inappropriate for fully automatic testing.

We might observe:

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

Problem and background:

- `QuickCheck` generates the lists `q` und `q'` randomly.
- Most of the generated pairs of lists will not be equivalent, and hence be discarded for the actual test.
- `QuickCheck` generates a maximum number of candidate arguments only (default: 1.000), and then stops, possibly before the number of 100 test cases is met.

Outlook

Enhancing usability of QuickCheck by adding support for

- Quantifying over subsets
  - by means of filters
  - by means of generators (type-based, weighted, size controlled,...)
- ...
- Test case monitoring

In the following:

- Illustrating this support by means of examples!

Chapter 5.4

Quantifying over Subsets

Background and Motivation

For `QuickCheck` holds:

- By default, parameters are quantified over the values of the underlying type
  (e.g., all integer lists)

Often, however, it is required:

- A quantification over subsets of these values
  (e.g., all sorted integer lists)

Quantifying over Subsets

`QuickCheck` offers several means for achieving this:

Representation of subsets in terms of

- Boolean functions that act as a filter for test cases
  - Adequate, if many elements of the underlying set are members of the relevant subset, too.
  - Inadequate, if only a few elements of the underlying set are members of the relevant subset.
- generators
  - A generator of type `Gen a` yields a random sequence of values of type `a`.
  - The property `forall set p` successively checks `p` on randomly generated elements of `set`.

Support by QuickCheck

For the effective usage of generators `QuickCheck` supports:

- different variants for the specification of relations such as `equiv`
  - As a Boolean function
    - easy to check equivalence of two values (but difficult to generate values that are equivalent).
  - As a function from a value to a set of related (e.g., equivalent) values (generator!)
    - easy to generate equivalent values (but difficult to check if two values are equivalent).

The latter option will be considered in more detail in the following chapter.

Chapter 5.5

Generating Test Data

Generators

The fundamental function to make a choice:

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

Note:

- The function `choose` generates "randomly" an element of the specified domain.
- `choose (1,n)` represents the set $\{1,...,n\}$.
- The type `Gen` is a monad (cp. Chapter 11).

Using choose

...we can define `equivQ`:

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

- Generates a random queue that contains the same elements as `q`.
- The number `k` of elements in the back of the queue will be chosen such that it is properly smaller than the total number of elements of the queue (under the assumption that the total number is different from `0`).

Application (1)

This allows us to check that

- generated elements are related, i.e., equivalent.

To this end check:

```
prop_EquivQ q = invariant q ==>
  forAll (equivQ q) $ \q' -> q 'equiv' q'
```

Note:

- Recall that $ means function application. Using $ allows the omission of parentheses (see the λ expression in the example).
- The property which is dual to prop_EquivQ, i.e., that all related elements can be generated, cannot be checked by testing.

Application (2)

This allows:

- Reformulating the property that addI maps equivalent queues to equivalent queues

```
prop_add_equiv q x = invariant q ==>
  forAll (equivQ q) $ \q' ->
              addI x q 'equiv' addI x q'
```

Remark:

- Other properties analogously

Next we consider: How to define generators.

Defining Generators

...is eased because of the monadic type of `Gen`.

It holds:

- `return a` always yields (generates) a and represents the singleton set $\{a\}$
- `do {x <- s; e}` can be considered the (generated) set $\{e \mid x \in s\}$

Type-based Generators (1)

...by means of the overloaded generator `arbitrary`, e.g. for the generation of arguments of properties:

Example 1:

```
prop_max_le x y = x <= x 'max' y
```

is equivalent to

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

Type-based Generators (2)

Example 2:

The set $\{y \mid y \geq x\}$ can be generated by

```
atLeast x = do diff <- arbitrary
               return (x + abs diff)
```

because of the equality

$$\{y \mid y \geq x\} = \{x + abs\ d \mid d \in \mathbb{Z}\}$$

that holds for numerical types.

Note: Similar definitions are possible for other types, too.

Selection

...between several generators can be achieved by means of a generator oneof that can be thought of as set union.

Example: Constructing a sorted list

```
orderedLists = do x <- arbitrary
                  listsFrom x
 where
  listsFrom x
   = oneof [return [], do y <- atLeast x
                          liftM (x:) (listsFrom y)]
```

Underlying intuition:

- A sorted list is either empty or the addition of a new head element to a sorted list of larger elements.

Weighted Selection (1)

- The `oneof` combinator picks with equal probability one of the alternatives.
- This often has an unduly impact on the test case generation (in the previous example the empty set will be selected too often).
- Remedy: A weight function `frequency` that assigns different weights to the alternatives.

```
frequency :: [(Int,Gen a)] -> Gen a
```

Weighted Selection (2)

Application:

```
listsFrom x
   = frequency [(1,return []),
                (4,do y <- atLeast x
                      liftM (x:) (listsFrom y)) ]
```

- A QuickCheck generator corresponds to a probability distribution over a set, not the set itself.
- The impact of the above assignment of weights is that on average the length of generated lists is 4.

The Type Class Arbitrary

If non-standard generators such as `orderedLists` are used frequently, it is advisable to make this type an instance of type class `Arbitrary`:

```
newtype OrderedList a = OL [a]

instance (Num a, Arbitrary a) =>
                Arbitrary (OrderedList a) where
  arbitrary = liftM OL orderedLists
```

Together with the re-definition of `insert` as

```
insert :: Ord a => a -> OrderedList a
                          -> OrderedList a
```

arguments generated for it will automatically be ordered.

Controlling the Size of Generated Test Data

- This is usually wise for type-based test data generation
- It is explicitly supported by QuickCheck

Controlling the Size of Generated Test Data

Generators that depend on the size can be defined by:

```
sized :: (Int -> Gen a) -> Gen a
            -- For defining size aware gen.

sized $ \n -> do len <- choose (0,n)
                 vector len -- Application of sized
                            -- in the Def. of the
                            -- default list generator

vector n = sequence [arbitrary | i <- [1..n]]
                            -- generates random list
                            -- of length n

resize :: Int -> Gen a -> Gen a
                            -- for controlling the size
                            -- of generated values

sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
                            -- Application of resize
```

Generators for User-defined Types

Test data generators for

- predefined ("built-in") types of Haskell
  - are provided by `QuickCheck`
  - for user-defined types, this is not possible
- user-defined types
  - have to be provided by the user in terms of defining a suitable instance of the type class `Arbitrary`
  - require usually, especially in case of recursive types, to control the size of generated test data

Example: Binary Trees (1)

Consider type (Tree a):

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

The following definition of the test-data generator is obvious:

```
instance Arbitrary a => Arbitrary (Tree a) where
 arbitrary =
  frequency [(1,return Leaf),
             (3,liftM3 Branch
                 arbitrary arbitrary arbitrary)]
```

Example: Binary Trees (2)

Note:

- The assignment of weights (1 vs. 3) has been done in order to avoid the generation of all too many trivial trees of size 1.
- Problem: The likelihood that a generator comes up with a finite tree, is only one third.

  $\rightsquigarrow$ this is because termination is possible only, if all subtrees generated are finite. With increasing breadth of the trees, the requirement of always selecting the "terminating" branch has to be satisfied at ever more places simultaneously.

Example: Binary Trees (3)

Remedy:

- Usage of the parameter `size` in order to ensure
  - termination and
  - “reasonable” size

  of the generated trees.

Example: Binary Trees (4)

Implementation:

```
instance Arbitrary a => Arbitrary (Tree a) where
 arbitrary = sized arbTree

arbTree 0 = return Leaf
arbTree n | n>0 =
 frequency [(1,return Leaf),
            (3,liftM3 Branch shrub arbitrary shrub)]
  where
   shrub = arbTree (n 'div' 2)
```

Note: `shrub` is a generator for small(er) trees.

Example: Binary Trees (5)

Remark:

- `shrub` is a generator for “small(er)” trees.
- `shrub` is not bounded to a special tree; the two occurrences of `shrub` will usually generate different trees.

- Since the size limit for subtrees is halved, the total size is bounded by the parameter `size`.
- Defining generators for recursive types must usually be handled specifically as in this example.

Chapter 5.6

Monitoring, Reporting, and Coverage

Test-Data Monitoring

In practice, it is useful

- to monitor the generated test cases in order to obtain a hint on the quality and the coverage of test cases

of a `QuickCheck` run.

For this purpose `QuickCheck` provides

- an array of monitoring and reporting possibilities.

Usefulness of Test-Data Monitoring

Why is test-data monitoring meaningful?

Reconsider the example of inserting into a sorted list:

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

Test-Data Monitoring and Test Coverage

QuickCheck performs the check of prop_InsertOrdered such that:

- lists are generated randomly
- each generated list will be checked, if it is sorted (used test case) or not (discarded test case)

Obviously, it holds:

- the likelihood that a randomly generated list is sorted is the higher the shorter the list is

This introduces the danger that

- the property prop_InsertOrdered is mostly tested with lists of length one or two
- even a successful test is not meaningful

Test-Data Monitoring using trivial (1)

For monitoring purposes QuickCheck provides a

- combinator trivial, where the meaning of "trivial" is user-definable.

Example:

```
prop_InsertOrdered :: Integer -> [Integer]
                                          -> Property
prop_InsertOrdered x xs = ordered xs ==>
  trivial (length xs <= 2) $ ordered (insert x xs)
```

with

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial)
```

Test-Data Monitoring using `trivial` (2)

Observation:

- 91% are too many trivial test cases in order to ensure that the total test is meaningful
- The operator ==> should be used with care in test-case generators

Remedy:

- User-defined generators
  $\rightsquigarrow$ as in the example of `prop_InsertOrdered` in Chapter 5.1 ("Second Fix (systematic)").

Test-Data Monitoring using `classify` (1)

The combinator `trivial` is

- instance of a more general combinator `classify`

  ```
  trivial p = classify p "trivial"
  ```

Test-Data Monitoring using classify (2)

Multiple applications of classify allow an even more refined test-case monitoring:

```
prop_InsertOrdered x xs = ordered xs =>
   classify (null xs) "empty lists" $
      classify (length xs == 1) "unit lists" $
         ordered (insert x xs)
```

This yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```

Test-Data Monitoring using collect

Going beyond, the combinator collect allows us to keep track on all test cases:

```
prop_InsertOrdered x xs = ordered xs =>
   collect (length xs) $ ordered (insert x xs)
```

This yields a histogram of values:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
5% 3.
```

Chapter 5.7

Implementation of QuickCheck

On the Implementation of QuickCheck (1)

A Glimpse into the implementation:

```
class Testable a where
  property :: a -> Property

newtype Property = Prop (Gen Result)

instance Testable Bool where
  property b = Prop (return (resultBool b))

instance (Arbitrary a, Show a, Testable b) =>
                          Testable (a->b) where
  property f = forAll arbitrary f

instance Testable Property where
  property p = p

quickCheck :: Testable a => a -> IO ()
```

On the Implementation of QuickCheck (2)

QuickCheck

- consists in total of about 300 lines of code.
- has initially been presented by Koen Claessen and John Hughes:

  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.

Summing up (1)

In general, it holds:

- Formalizing specifications is meaningful (even without a subsequent formal proof of soundness).

Experience shows:

- Specifications provided are often (initially) faulty themselves.

Summing up (2)

QuickCheck is an effective tool

- to disclose bugs in
  - programs and
  - specifications

  with little effort.
- to reduce
  - test costs

  while simultaneously
  - testing more thoroughly.

Summing up (3)

Investigations of Richard Hamlet

- Richard Hamlet. Random Testing. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994

indicate that

- a high number of test cases yields meaningful results even in the case of random testing.

Moreover

- The generation of random test cases is often "cheap."

Hence, there are many reasons advising

- the routine usage of a tool like QuickCheck!

Summing up (4)

Besides QuickCheck there are various other combinator libraries supporting the lightweight testing of Haskell programs, e.g.:

- EasyCheck
- SmallCheck
- Lazy SmallCheck
- Hat

Summing up (5)

The presentation of this chapter is closely based on:

- Koen Claessen, John Hughes. Specification-based Testing with QuickCheck. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 17-39, 2003.

For implementation details and applications refer 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.

Chapter 5: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18.2, QuickCheck)
- 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.
- 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.

Chapter 5: Further Reading (2)

- 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)

- 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`)
- 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)

Chapter 6

Verification

Motivation

Though often amazingly effective, testing is limited to

- showing the presence of errors.
  It can not show their absence!

By contrast, verification is able to

- proving the absence of errors!

In this chapter

...we will consider important proof techniques for verifying properties of functional (and other) programs that may operate on

- elementary data such as
  - integers
  - strings
  - ...
- composed data (in Haskell: algebraic data types) such as
  - trees
  - lists (which are finite by definition)
  - streams (which are infinite by definition)
  - ...

Outline of the Proof Techniques

We already considered (cf. Chapter 4):

- Equational reasoning

We will consider in this chapter:

- Basic inductive proof principles
  - Natural (or mathematical) induction
  - Strong induction
  - Structural induction
- Specialized inductive proof principles
  - Induction on lists
  - Induction on streams
- Coinduction
- Fixed point induction

Before going into details (1)

...it is worth noting:

Though of different power, testing and verification are both instances of approaches that aim at

- ensuring the correctness of a program or system.

Before going into details (2)

Conceptually, we can distinguish between approaches that strive for ensuring correctness by

- Construction
  $\rightsquigarrow$ applied a priori/on-the-fly of the program development
- Checking
  $\rightsquigarrow$ applied a posteriori of the program development
  - Verification
  - Testing (only to a limited extent if not exhaustive)

Before going into details (3)

With this in mind, we may loosely conclude:

- Correctness by Construction
  - Equational Reasoning
- Correctness by Checking
  - Verification
  - Testing

Chapter 6.1

Equational Reasoning – Correctness by Construction

Equational Reasoning

...is sometimes also called

- proof by program calculation.

It has been considered and demonstrated previously. Consider Chapter 4 for details.

Chapter 6.1: Further Reading (1)

- 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.
- 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.

Chapter 6.1: Further Reading (2)

- 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 6.2

Basic Inductive Proof Principles

Outline

Basic inductive proof principles are:

- Natural or mathematical induction (dtsch. vollständige Induktion)
- Strong induction (dtsch. verallgemeinerte Induktion)
- Structural induction (dtsch. strukturelle Induktion)

Basic Inductive Proof Principles

Let P be a property; let S be a set of values s that are (inductively) constructed from a set of (structurally simpler) values $subs(s)$; let $\mathbb{N}$ denote the set of natural numbers.

The principles of

- Natural (mathematical) induction

  $$(P(1) \wedge (\forall n \in \mathbb{N}.\, P(n) \Rightarrow P(n{+}1))) \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\ P(n)$$

- Strong induction

  $$(\forall n \in \mathbb{N}.\, (\forall m < n.\, P(m)) \Rightarrow P(n)) \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\ P(n)$$

- Structural induction

  $$(\forall\, s \in S.\forall\, s' \in subs(s).\, P(s')) \Rightarrow P(s)) \;\Rightarrow\; \forall\, s \in S.\ P(s)$$

Note

The proof principles of

- natural (mathematical)
- strong
- structural

induction are equally expressive and powerful.

Illustrating Examples

Next we provide some typical examples illustrating the usage of these three basic inductive principles of

- natural (mathematical)
- strong
- structural

induction.

Chapter 6.2.1

Natural Induction

Example A

Theorem (6.2.1.1)

$$\forall\, n \in I\!N.\ \sum_{i=1}^{n} i = \frac{n * (n+1)}{2}$$

Proof: By means of natural (mathematical) induction.

Proof of Theorem 6.2.1.1 (1)

Base case: $n = 1$. In this case we obtain the desired equality by a straightforward calculation:

$$\begin{aligned}
\sum_{i=1}^{n} i &= \sum_{i=1}^{1} i \\
&= 1 \\
&= \frac{2}{2} \\
&= \frac{1 * 2}{2} \\
&= \frac{1 * (1+1)}{2} = \frac{n * (n+1)}{2}
\end{aligned}$$

Proof of Theorem 6.2.1.1 (2)

Inductive case: Applying the induction hypothesis (IH) once, we obtain as desired:

$$
\begin{aligned}
\sum_{i=1}^{n+1} i &= (n+1) + \sum_{i=1}^{n} i \\
\text{(IH)} &= (n+1) + \frac{n * (n+1)}{2} \\
&= (n+1) * (\frac{n}{2} + 1) \\
&= \frac{(n+1) * (n+2)}{2} = \frac{(n+1) * ((n+1)+1)}{2}
\end{aligned}
$$

□

Example B

Theorem (6.2.1.2)

$$\forall\, n \in I\!N.\ \sum_{i=1}^{n} (2 * i - 1) = n^2$$

Proof: By means of natural (mathematical) induction.

Proof of Theorem 6.2.1.2 (1)

Base case: $n = 1$. In this case we obtain the desired equality by a straightforward calculation:

$$\begin{aligned}\sum_{i=1}^{n}(2*i-1) &= \sum_{i=1}^{1}(2*i-1)\\ &= 2*1-1\\ &= 2-1\\ &= 1\\ &= 1^2 = n^2\end{aligned}$$

Proof of Theorem 6.2.1.2 (2)

Inductive case: Applying the induction hypothesis (IH) once, we obtain as desired:

$$
\begin{array}{rcl}
\sum_{i=1}^{n+1}(2*i-1) & = & 2*(n+1)-1+\sum_{i=1}^{n}(2*i-1) \\
\text{(IH)} & = & (2*(n+1)-1)+n^2 \\
 & = & 2n+2-1+n^2 \\
 & = & 2n+1+n^2 \\
 & = & n^2+2n+1 \\
 & = & n^2+n+n+1 \\
 & = & (n+1)*(n+1)=(n+1)^2
\end{array}
$$

□

Chapter 6.2.2

Strong Induction

Fibonacci Function

The Fibonacci function is defined by:

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

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

Example

Theorem (6.2.2.1)

$$\forall\, n \in I\!N_0.\ fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Proof: By means of strong induction.

Key Idea for proving Theorem 6.2.2.1

Using the induction hypothesis (IH) that for all $k < n$, $n \in \mathbb{N}_0$, the equality

$$fib(k) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k}{\sqrt{5}}$$

holds, we can prove the premise underlying the implication of the principle of strong induction for all natural numbers n by investigating the following basic and inductive cases.

Proof of Theorem 6.2.2.1 (2)

Base case 1: $n = 0$. In this case, a straightforward calculation yields the desired equality:

$$fib(0) = 0 = \frac{0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}}$$

Base case 2: $n = 1$. Again, a straightforward calculation yields as desired:

$$fib(1) = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\frac{1}{2} + \frac{\sqrt{5}}{2} - (\frac{1}{2} - \frac{\sqrt{5}}{2})}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}}$$

Proof of Theorem 6.2.2.1 (3)

Inductive case: $n \geq 2$. Applying the IH for $n-2, n-1$ yields as desired:

$$fib(n) = fib(n-2) + fib(n-1)$$

$$\text{(2x IH)} \quad = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}}$$

$$= \frac{\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\right] - \left[\left(\frac{1-\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]}{\sqrt{5}}$$

$$= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1+\sqrt{5}}{2}\right] - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1-\sqrt{5}}{2}\right]}{\sqrt{5}}$$

$$(*) \quad = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left(\frac{1+\sqrt{5}}{2}\right)^{2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left(\frac{1-\sqrt{5}}{2}\right)^{2}}{\sqrt{5}}$$

$$= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}}{\sqrt{5}}$$

Proof of Theorem 6.2.2.1 (4)

The equality marked by $(*)$ follows from the below two calculations that make use of the binomial formulae.

We have:

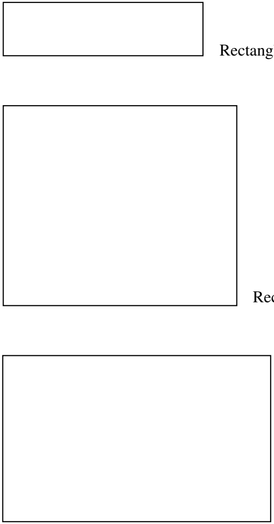
$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = 1+\frac{1+\sqrt{5}}{2}$$

Similarly we get:

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = 1+\frac{1-\sqrt{5}}{2}$$

□

Excursus: Which Rectangle looks ‘nicest’?

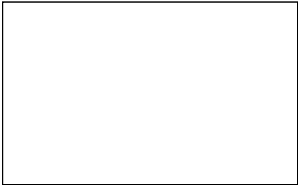


Rectangle 1

Rectangle 2

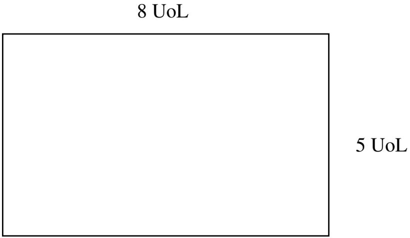
Rectangle 3

Most People say ‘Rectangle 3’!



Rectangle 3

Why?

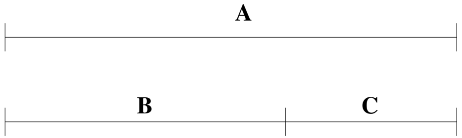


$$8\ UoL/5\ UoL = 1.6$$

The value 1.6 comes close to

...the Golden Ratio:

$$\phi =_{df} \frac{1+\sqrt{5}}{2} = 1.61803398874989...$$



Intuitively:

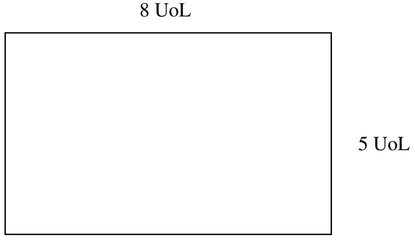
- The ratio of section A and section B is the same as the ratio of section B and section C

$$A/B = B/C$$

The value of this ratio is denoted by ϕ.

The Golden Ratio

...is perceived as harmonious:



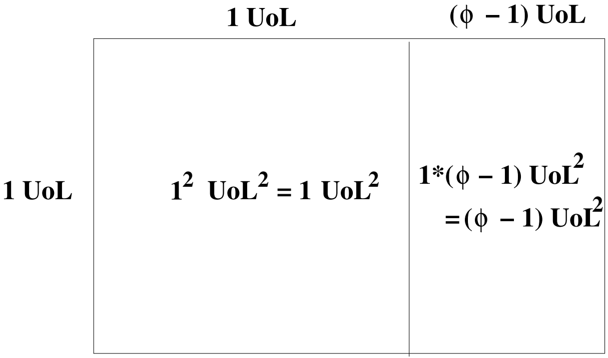
$$8\ UoL/5\ UoL = 1.6$$

What is the value of ϕ?

$$\begin{aligned}
& \frac{x+1}{x} && = \frac{x}{1} = \phi \\
\iff \quad & 1 + \frac{1}{x} && = x = \phi \\
\text{Thus:} \quad & 1 + \frac{1}{\phi} && = \phi \\
\Rightarrow \quad & \phi + 1 && = \phi^2 \\
\Rightarrow \quad & \phi^2 - \phi - 1 && = 0 \\
\Rightarrow \quad & \phi = \frac{1+\sqrt{5}}{2} = 1.618\ldots \\
& (\ \phi' = \frac{1-\sqrt{5}}{2} = -0.618\ldots\ )
\end{aligned}$$

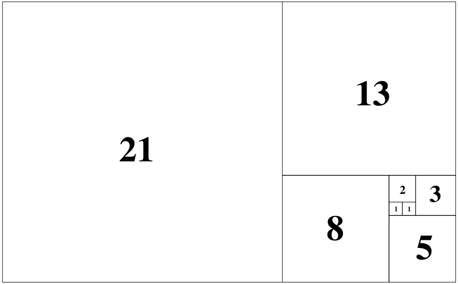
The Golden Ratio

...not only in terms of the ratios of sections but also in terms of the ratios of the areas of e.g. rectangles:



The Golden Ratio and Fibonacci Numbers (1)

The Golden Ratio, rectangles and the Fibonacci numbers:



The Golden Ratio and Fibonacci Numbers (2)

The sequence of Fibonacci numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

The sequence of the ratios of the Fibonacci numbers:

$$\begin{aligned}
1/1 &= 1 \\
2/1 &= 2 \\
3/2 &= 1.5 \\
5/3 &= 1.\overline{6} \\
8/5 &= 1.6 \\
13/8 &= 1.625 \\
21/13 &= 1.615384615384615 \\
34/21 &= 1.619047619047619 \\
\ldots & \\
1,346,269/832,040 &= 1.618033988750541
\end{aligned}$$

The Golden Ratio and Fibonacci Numbers (3)

...as the limit of the ratios of the Fibonacci numbers.

We have:

$$\lim_{n \to \infty} \frac{fib(n+1)}{fib(n)} = \frac{1+\sqrt{5}}{2} = \phi$$

Chapter 6.2.3

Structural Induction

Arithmetic Expressions

The set $\mathcal{AE}$ of (simple) arithmetic expressions is defined by the BNF rule:

$$e \; ::= \; n \mid v \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 * e_2) \mid (e_1 / e_2)$$

where n and v stand for an (integer) numeral and variable, respectively.

Example A

Theorem (6.2.3.1)

Let $e \in \mathcal{AE}$, let lp_e and rp_e denote the number of left and right parentheses of e. Then we have:

$$lp_e = rp_e$$

Proof: By means of structural induction.

Proof of Theorem 6.2.3.1 (1)

Base case 1: Let $e \equiv n$, n a numeral.

In this case e does not contain any parentheses. This means $lp_e = 0 = rp_e$ which yields the desired equality of lp_e and rp_e.

Base case 2: Let $e \equiv v$, v a variable.

As above, we conclude $lp_e = 0 = rp_e$ obtaining the desired equality of lp_e and rp_e also in this case.

Proof of Theorem 6.2.3.1 (2)

Inductive case: Let $e_1, e_2 \in \mathcal{AE}$, let $\circ \in \{+, -, *, /\}$, and let $e \equiv (e_1 \circ e_2)$.

By means of the induction hypothesis (IH) we can assume $lp_{e_1} = rp_{e_1}$ and $lp_{e_2} = rp_{e_2}$. This allows us to prove the desired equality of lp_e and rp_e thereby completing the proof as follows:

$$\begin{array}{rcl} & & lp_e \\ (\, e \equiv (e_1 \circ e_2)\,) & = & lp_{(e_1 \circ e_2)} \\ & = & 1 + lp_{e_1} + lp_{e_2} \\ (2\text{x IH}) & = & rp_{e_1} + rp_{e_2} + 1 \\ & = & rp_{(e_1 \circ e_2)} \\ (\, e \equiv e_1 \circ e_2\,) & = & rp_e \end{array}$$

□

Example B

Theorem (6.2.3.2)

Let $e \in \mathcal{AE}$, let p_e and op_e denote the number of parentheses and of operators of e, respectively. Then we have:

$$p_e = 2 * op_e$$

Proof: By means of structural induction.

Proof of Theorem 6.2.3.2 (1)

Base case 1: Let $e \equiv n$, n a numeral.

In this case e does not contain any parentheses or operators. This means $p_e = 0 = op_e$, which yields as desired

$$p_e = 0 = 2 * 0 = 2 * op_e$$

Base case 2: Let $e \equiv v$, v a variable.

As above, we conclude $p_e = 0 = op_e$ obtaining the desired equality

$$p_e = 0 = 2 * 0 = 2 * op_e$$

in this case, too.

Proof of Theorem 6.2.3.2 (2)

Inductive case: Let $e_1, e_2 \in \mathcal{AE}$, let $\circ \in \{+, -, *, /\}$, and let $e \equiv (e_1 \circ e_2)$.

By means of the induction hypothesis (IH) we can assume that $p_{e_1} = 2 * op_{e_1}$ and $p_{e_2} = 2 * op_{e_2}$. With these equalities we obtain as desired:

$$
\begin{array}{rcl}
 & & p_e \\
(\, e \equiv (e_1 \circ e_2)\,) & = & p_{(e_1 \circ e_2)} \\
 & = & 1 + p_{e_1} + p_{e_2} + 1 \\
(2\text{x IH}) & = & 2 * op_{e_1} + 2 + 2 * op_{e_2} \\
 & = & 2 * op_{e_1} + 2 * 1 + 2 * op_{e_2} \\
 & = & 2 * (\, op_{e_1} + 1 + op_{e_2}\,) \\
 & = & 2 * op_{(e_1 \circ e_2)} \\
(\, e \equiv (e_1 \circ e_2)\,) & = & 2 * op_e
\end{array}
$$

□

Example C

Theorem (6.2.3.3)

Let $e \in \mathcal{AE}$ be an arithmetic expression of depth n, let opd_e denote the number of of operands of e. Then we have:

$$opd_e \leq 2^n$$

Proof: By means of structural induction.

Proof of Theorem 6.2.3.3 (1)

Base case 1: Let $e \equiv n$, n a numeral.

In this case e has depth 0 and contains 1 operand. This yields as desired:

$$opd_e = opd_n = 1 = 2^0 \leq 2^0$$

Base case 2: Let $e \equiv v$, v a variable.

As in the previous case e has depth 0 and contains 1 operand. Again we obtain as desired:

$$opd_e = opd_v = 1 = 2^0 \leq 2^0$$

Proof of Theorem 6.2.3.3 (2)

Inductive case: Let $e_1, e_2 \in \mathcal{AE}$ be arithmetic expressions of depth n and m, respectively. Without losing generality let $m \leq n$. Let $\circ \in \{+, -, *, /\}$, and let $e \equiv (e_1 \circ e_2)$.

In this case expression e has depth $n+1$. By means of the induction hypothesis (IH) we can assume $opd_{e_1} \leq 2^n$ and $opd_{e_2} \leq 2^m$. Using these inequalities the proof can be completed as follows:

$$
\begin{array}{rcl}
 & & opd_e \\
(\, e \equiv (e_1 \circ e_2)\,) & = & opd_{(e_1 \circ e_2)} \\
 & = & opd_{e_1} + opd_{e_2} \\
\text{(2x IH)} & \leq & 2^n + 2^m \\
(m \leq n) & \leq & 2^n + 2^n \\
 & = & 2 * 2^n \\
 & = & 2^{n+1}
\end{array}
$$

□

Chapter 6.3

Inductive Proofs on Algebraic Data Types

Chapter 6.3.1

Induction and Recursion

Induction and Recursion

...are closely related.

Intuitively:

- Induction describes things starting from something very simple, and building up from there: It is a bottom-up principle.
- Recursion starts from the whole thing, working backward to the simple case(s): It is a top-down principle.

Thus:

- Induction (bottom-up) and recursion (top-down) can be considered the two sides of the same coin.

In fact

The preferred usage of

- induction over recursion in some contexts (e.g., defining data structures) resp. vice versa in others (e.g., defining algorithms) is often mostly due to historical reasons.

Data types:

```
data Tree = Leaf Integer
          | Node Tree Tree
```

Algorithms:

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n-1)
```

Examples

- 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 merge sort:

  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.

Summing up

- Definitions of data structures follow often an inductive definition pattern, e.g.:
  - A list is either empty or a pair consisting of an element and another list.
  - A tree is either empty or consists of a node and a set of subtrees.
  - An arithmetic expression is either a numeral or a variable, or is composed of (two) arithmetic expressions by means of a (binary) arithmetic operator.
- Algorithms (functions) on data structures follow often a recursive definition pattern, e.g.:
  - The function `length` computing the length of a list.
  - The function `depth` computing the depth of a tree.
  - The function `evaluate` computing the value of an arithmetic expression (given a valuation of its variables).

Chapter 6.3.2

Inductive Proofs on Trees

Inductive Proofs on Trees

Let

```
data Tree = Leaf Integer
          | Node Tree Tree
```

Theorem (6.3.2.1)

Let t be a value, i.e., a tree, of type Tree of depth n, let leaves(t) denote the number of leafs of t.
Then we have:

$$leaves(t) \leq 2^n$$

Proof: By means of structural induction.

Proof of Theorem 6.3.2.1 (1)

Base case: Let $\texttt{t} \equiv \texttt{Leaf k}$ for some integer $\texttt{k}$.

In this case $\texttt{t}$ has depth 0 and contains 1 leaf. This yields as desired:

$$\texttt{leaves(t)} = \texttt{leaves(Leaf k)} = 1 = 2^0 \leq 2^0$$

Proof of Theorem 6.3.2.1 (2)

Inductive case: Let t1 and t2 be two values of type Tree of depth n and m, respectively. Without losing generality let $m \leq n$, and let $\texttt{t} \equiv \texttt{Node t1 t2}$.

In this case t is a tree of depth $n+1$. By means of the inductive hypothesis (IH) we can assume $\texttt{leaves(t1)} \leq 2^n$ and $\texttt{leaves(t2)} \leq 2^m$. Using these inequalities the proof can be completed as follows:

$$
\begin{array}{rcl}
 & & \texttt{leaves(t)} \\
(\texttt{t} \equiv \texttt{Node t1 t2}) & = & \texttt{leaves(Node t1 t2)} \\
 & = & \texttt{leaves(t1)} + \texttt{leaves(t2)} \\
(\text{2x IH}) & \leq & 2^n + 2^m \\
(m \leq n) & \leq & 2^n + 2^n \\
 & = & 2 * 2^n \\
 & = & 2^{n+1}
\end{array}
$$

□

Chapter 6.3.3

Inductive Proofs on Lists

Preliminaries

Recall:

- A list is by definition finite.

Given a list, it is called

- partial, if it is built from the undefined list
- defined, if it is not partial and all its elements are defined

Note:

- For inductively proving properties on lists we have to distinguish the two cases of
  - defined lists (cf. Chapter 6.3.3)
  - partial lists with possibly undefined lists (cf. Chapter 6.3.4)

Inductive Proofs on Defined Lists

The inductive proof pattern for defined lists:

Let P be a property on lists.

1. Base case: Prove that P is true for the empty list, i.e. prove $P([\,])$.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x : xs)$ is true (induction step).

Note:

- The above pattern is an instance of the more general pattern of structural induction.
- A property P proved using this pattern is true for lists with only defined elements of any finite length.

Example A: Induction on Lists (1)

Let

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Lemma (6.3.3.1)

For all defined lists xs, ys holds:

$$length\ (xs ++ ys) = length\ xs\ +\ length\ ys$$

Proof by induction on the structure of *xs*.

Example A: Induction on Lists (2)

Base case:

$$
\begin{aligned}
& length([\,] +\!\!+ ys) \\
= \quad & length\ ys \\
= \quad & 0 + length\ ys \\
= \quad & length\ [\,] + length\ ys
\end{aligned}
$$

Inductive case:

$$
\begin{aligned}
& length((x : xs) +\!\!+ ys) \\
= \quad & length\ (x : (xs +\!\!+ ys)) \\
= \quad & 1 + length\ (xs +\!\!+ ys) \\
\text{(IH)} \quad = \quad & 1 + (length\ xs + length\ ys) \\
= \quad & (1 + length\ xs) + length\ ys \\
= \quad & length\ (x : xs) + length\ ys
\end{aligned}
$$

□

Example B: Induction on Lists (1)

Let

```
listSum :: Num a => [a] -> a
listSum []     = 0
listSum (x:xs) = x + listSum xs
```

Lemma (6.3.3.2)

For all defined lists xs holds:

$$listSum\ xs \;=\; foldr\ (+)\ 0\ xs$$

Proof by induction on the structure of *xs*.

Example B: Induction on Lists (2)

Base case:

$$\begin{aligned} & \mathit{listSum}\ [\,] \\ = \ & 0 \\ = \ & \mathit{foldr}\ (+)\ 0\ [\,] \end{aligned}$$

Inductive case:

$$\begin{aligned} & \mathit{listSum}\ (x : xs) \\ = \ & x + \mathit{listSum}\ xs \\ \text{(IH)} \quad = \ & x + \mathit{foldr}\ (+)\ 0\ xs \\ = \ & \mathit{foldr}\ (+)\ 0\ (x : xs) \end{aligned}$$

□

Example C: Induction on Lists w/ map (1)

Properties of map that can be proved by induction on lists.

```
map (f.g)          = map f . map g
map f.tail         = tail . map f
map f . reverse = reverse . map f
map f . concat  = concat . map (map f)
map f (xs++ys)  = map f xs ++ map f ys

map (\x -> x)   = \y -> y
```

```
Note: \x -> x :: a -> a
      \y -> y :: [a] -> [a]
```

Example C: Induction on Lists w/ `map` (2)

Lemma (6.3.3.3)

If f is strict, it is true:

```
f . head = head . map f
```

Proof by induction on the structure of lists.

Example C: Induction on Lists w/ map (3)

Base case:

$$
\begin{array}{rcl}
 & & (f \;.\; head)\; [\,] \\
\text{(Def. of “.”)} & = & f\; (head\; [\,]) \\
 & = & f\; \bot \\
(f \text{ strict}) & = & \bot \\
 & = & head\; [\,] \\
\text{(Def. of } map\text{)} & = & head\; (map\; f\; [\,]) \\
\text{(Def. of “.”)} & = & (head\; .\; map\; f)\; [\,]
\end{array}
$$

Inductive case:

$$
\begin{array}{rcl}
 & & f\; .\; head\; (x : xs) \\
\text{(Def. of “.”)} & = & f\; (head\; (x : xs)) \\
 & = & f\; x \\
 & = & head\; (f\; x\; :\; map\; f\; xs) \\
\text{(Def. of } map\text{)} & = & head\; (map\; f\; (x : xs)) \\
\text{(Def. of “.”)} & = & (head\; .\; map\; f)\; (x : xs)
\end{array}
$$

□

Example D: Induction on Lists w/ `fold`

Properties of `fold` that can be proved by induction on lists.

- If `op` is associative with `e 'op' x = x` and `x 'op' e = x` for all `x`, then for all finite `xs`

  `foldr op e xs = foldl op e xs`

  is true.
- If `x 'op1' (y 'op2' z) = (x 'op1' y) 'op2' z` and `x 'op1' e = e 'op2' x`, then for all finite `xs`

  `foldr op1 e xs = foldl op2 e xs`

  is true.
- For all finite `xs`

  `foldr op e xs = foldl (flip op) e (reverse xs)`

  is true.

Example D: Induction on Lists w/ (++)

Properties of (++) that can be proved by induction on lists.

- For all xs, ys, and zs it is true:

```
(xs++ys) ++ zs = xs ++ (ys++zs)
                   (Associativity of (++))

xs ++ []       = [] ++ xs
                   ([] is neutral element of (++))
```

Example E: Induction on Lists w/ take/drop

Properties of take and drop that can be proved by induction on lists.

- For all m, n with m,n $\geq$ 0 and finite xs it is true:

```
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)
```

- If n $\geq$ m, it is additionally true:

```
drop m . take n = take (n-m) . drop m
```

Example F: Induction on Lists w/ `reverse`

Properties of `reverse` that can be proved by induction on lists.

- For all finite `xs` it is true:

```
head (reverse xs) = last xs
last (reverse xs) = head xs
```

- For all finite `xs` with only defined elements it is true:

```
reverse (reverse xs) = xs
```

Chapter 6.3.4

Inductive Proofs on Partial Lists

Preliminaries

Computations that

- fail to terminate
- are faulty, i.e., produce an error

do not give a proper, i.e., a defined value.

The value of such computations is called the

- undefined value.

The undefined value is usually denoted by $\perp$ ("bottom").

Examples

The function

```
buggy_fac :: Integer -> Integer
buggy_fac n = (n-1) * buggy_fac n
buggy_fac 0 = 1
```

...induces for every argument a non-terminating computation.

The function

```
buggy_div :: Integer -> Integer
buggy_div n = div n 0
```

...produces an error for each argument called with.

The Undefined Value in Haskell

The undefined value $\bot$

- is an element of every data type of Haskell representing the value of a
  - faulty or non-terminating computation.

  $\bot$ can be considered the "least accurate" approximation of (ordinary) values of the corresponding data type.

The definition

- Polymorphic

  ```
  bottom :: a
  bottom = bottom
  ```

  Concrete

  ```
  bottom :: Integer
  bottom = bottom
  ```

  is the most simple expression (of arbitrary type) whose evaluation leads to a non-terminating computation with value $\bot$.

The Undefined Value and Lists

The undefined value ⊥ may occur as

- an "ordinary" element of a list
- a list itself.

Example:

```
lst1 = 2 : bottom : 5 : 7 : []
lst2 = 2 : 3 : 5 : 7 : bottom
lst3 = 2 : bottom : 5 : 7 : bottom
lst4 = 2 : 3 : 5 : 7 : []
```

Note:

- The occurrence of `bottom` in `lst1` and the first occurrence of `bottom` in `lst3` are of type `Integer`.
- The occurrence of `bottom` in `lst2` and the second occurrence of `bottom` in `lst3` are of type `[Integer]`.

Defined and Partial Lists

Definition (6.3.4.1, Defined Values)

A value of a data type is defined, if it is not equal to $\perp$.

Definition (6.3.4.2, Defined and Partial Lists)

A list is

- defined, if it is a list of defined values
- partial, if it is built from the undefined list, i.e., if its tail is the undefined list $\perp$

Example:

- `lst4` is a defined list, while `lst1`, `lst2`, `lst3` are not.
- `lst2` and `lst3` are partial, while `lst1` is neither defined nor partial (note: `lst1` contains an undefined element but is not built from the undefined list).

Examples of Partial Lists

Successively increasingly defined partial lists:

- `bottom` (*the undefined list, i.e., the "least defined" partial list*)
- `1 : bottom` (*partial list*)
- `1 : 2 : bottom` (*partial list*)
- `1 : 2 : 3 : bottom` (*partial list*)
- ...
- `1 : 2 : 3 : 4 : 5 : 6 : 7 : bottom` (*partial list*)
- ...

Properties of Functions on Partial Lists (1)

```
reverse (lst1) ->> [9,7  ...followed by an infinite wait
reverse (lst2) ->> ...infinite wait
reverse (lst3) ->> ...infinite wait
reverse (lst4) ->> [7,5,3,2]

head (tail (reverse lst1)) ->> 7
head (tail (tail (reverse lst1)]) ->> ...infinite wait
last (lst1) ->> 7

last (lst2) ->> ...infinite wait
head (tail (reverse lst2)) ->> ...infinite wait

head (reverse lst1) ->> 9
head (tail (reverse lst1)) ->> 7
head (reverse (reverse lst1)) ->> 2
reverse (reverse (lst1) ->> [2 ...followed by an
                                infinite wait
```

Properties of Functions on Partial Lists (2)

```
length lst1 ->> 4
length lst2 ->> ...infinite wait
length lst3 ->> ...infinite wait
length lst4 ->> 4

length (take 3 lst1) ->> 3
length (take 2 lst2) ->> 2
length (take 3 lst3) ->> 3

length (take 4 lst4) ->> 4
length (take 5 lst4) ->> 4
length (take 4 lst2) ->> 4
length (take 5 lst2) ->> ...infinite wait
```

Properties of Functions on Partial Lists (3)

For understanding the different behaviours recall the definitions of `length` and `reverse`:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs -- x is not evaluated!
```

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs -- x is not evaluated!
```

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

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []      -- x is evaluated!
```

Inductive Proofs on Lists Reconsidered

The inductive proof pattern introduced at the beginning of Chapter 6.3.3 holds for

- defined lists.

For inductive proofs of properties on partial lists (such as `lst2`) with possibly undefined elements (such as `lst3`) it has to be replaced by the inductive proof principle shown next.

Inductive Proofs on Partial Lists w/ Possibly Undefined Elements

Inductive proof pattern for partial lists with possibly undefined elements:

Let P be a property on lists.

1. Base case: Prove that P is true for the empty list and for the undefined list, i.e. prove $P([])$ and $P(\bot)$.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x : xs)$ is true, for x being a defined and an undefined value (induction step).

Chapter 6.3.5

Inductive Proofs on Streams

Approximating Lists and Streams

Lists and Streams

- can be approximated by sequences of increasingly more accurate partial lists, also called approximants.

Approximating Lists by Partial Lists

The list

```
[1,2,3,4,5] = 1 : 2 : 3 : 4 : 5 : []
```

is approximated by the below sequence of partial lists that are increasingly more accurate approximations and ultimately culminate in the list [1,2,3,4,5]:

```
bottom
 1 : bottom
 1 : 2 : bottom
 1 : 2 : 3 : bottom
 1 : 2 : 3 : 4 : bottom
 1 : 2 : 3 : 4 : 5 : bottom
 1 : 2 : 3 : 4 : 5 : []
```

Approximating Streams by Partial Lists (1)

The stream

```
[1,2,3,4,5..]
```

of natural numbers is the limit of the infinite sequence of increasing approximations of partial lists:

```
bottom
1 : bottom
1 : 2 : bottom
1 : 2 : 3 : bottom
1 : 2 : 3 : 4 : bottom
1 : 2 : 3 : 4 : 5 : bottom
...
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : bottom
...
```

Approximating Streams by Partial Lists (2)

Note:

- Considering partial lists approximations of streams reminds to the strategy of partially outputting/printing streams by hitting `Ctrl-C` after some period of time.
- Extending this period of time further and further yields successively more accurate approximations of the stream.

Equality of Lists and Streams

Definition (6.3.5.1, Equality of Lists)

Two lists `xs` and `ys` are equal, if all their approximants are equal, i.e., if for all natural numbers `n`, `take n xs = take n ys`.

Definition (6.3.5.2, Infinite Lists, Streams)

A list `xs` is infinite or a stream, if for all natural numbers `n`, `take n xs /= take (n+1) xs`.

Definition (6.3.5.3, Equality of Streams)

Two streams `xs` and `ys` are equal, if for all natural numbers `n`, `xs!!n = ys!!n`.

Extending Properties from Lists to Streams

Properties on lists

- can be extensible to streams, e.g.,
  `take n xs ++ drop n xs = xs`
- but need not be extensible to streams, e.g.,
  `reverse (reverse xs)) = xs`

Similarly, properties that are true for every partial list of an approximating sequence of partial lists

- can be true for their limit
- but need not be true for their limit, e.g., "this list is partial".

Hence

Proving properties on streams thus demands for tailored

- proof strategies

that avoid such anomalies and paradoxes.

Fortunately

- The restriction "expressed as an equation in Haskell" is sufficient to ensure that a property that is true for every partial list of an approximating sequence is also true for its limit.

Inductive Proofs on Streams

Inductive proof pattern for streams with only defined elements:
Let P be a property on streams expressed as an equation in Haskell.

1. Base case: Prove that P holds for the least defined list, i.e. prove $P(\perp)$ (instead of $P([\,])$).
2. Inductive case: Assume that $P(xs)$ is true (induction hypothesis) and prove that $P(x : xs)$ is true (induction step).

Example A: Induction on Streams (1)

Lemma (6.3.5.4)

For all streams xs is true:

$$take\ n\ xs ++ drop\ n\ xs = xs$$

Proof by induction on the structure of `xs`.

Example A: Induction on Streams (2)

Base case:

$$\begin{aligned} & take\ n\ \bot +\!\!+\ drop\ n\ \bot \\ = \quad & \bot\ +\!\!+\ drop\ n\ \bot \\ = \quad & \bot \end{aligned}$$

Inductive case:

$$\begin{aligned} & take\ n\ (x : xs)\ +\!\!+\ drop\ n\ (x : xs) \\ = \quad & x\ :\ (take\ (n-1)\ xs\ +\!\!+\ drop\ (n-1)\ xs) \\ \text{(IH)} \quad = \quad & x\ :\ xs \end{aligned}$$

$\square$

Chapter 6.4

Approximation

Proof by Approximation

...is an important principle

- for proving properties of infinite objects, e.g. equality of streams
- has been applied in Chapter 6.3.5.
- is more general than the usage suggested there.

...will be considered in more detail in this chapter.

Preliminaries

Definition (6.4.1, Partially Ordered Set)

A relation R on M is called a partially ordered set (or partial order) iff R is reflexive, transitive, and anti-symmetric.

Definition (6.4.2, Chain)

Let $(P, \sqsubseteq)$ be a partially ordered set. A subset $C \subseteq P$ is called a chain of P, if the elements of C are totally ordered.

Remark

- Refer to Appendix to recall the meaning of terms if necessary.

Domains

Definition (6.4.3, Domain)

A set D with a partial order $\sqsubseteq$ is called a domain, if

1. D has a least element $\bot$
2. $\bigsqcup C$ exists for every chain C in D

Example

- Let $\mathcal{P}(\mathbb{N})$ denote the power set of $\mathbb{N}$. Then $(\mathcal{P}(\mathbb{N}), \sqsubseteq)$ with $\sqsubseteq =_{df} \subseteq$ is a domain with least element $\emptyset$ and $\bigsqcup C = \bigcup C$ for every chain C in $\mathcal{P}(\mathbb{N})$.

Note

- A domain is a (chain) complete partial order (cf. Appendix)
- The relation $\sqsubseteq$ of a domain is also called approximation order.

Approximation Order for Lists and Streams

Definition (6.4.4, Approximation Order)

We define the following relation on lists and streams:

$$\begin{array}{lcll} \bot & \sqsubseteq & xs & \\ [\,] & \sqsubseteq & xs & =_{df} \; xs = [\,] \\ x : xs & \sqsubseteq & y : ys & =_{df} \; x \sqsubseteq y \;\wedge\; xs \sqsubseteq ys \end{array}$$

Lemma (6.4.5, Domain Property of List Types)

Let a be a type such that its values form a domain. Then the values of the data types [a] form under the approximation order of Definition 6.4.4 a domain.

Approximating Lists by Partial Lists

By means of Definition 6.4.4, we have:

$$\bot \sqsubseteq x_0 : \bot \sqsubseteq x_0 : x_1 : \bot \sqsubseteq x_0 : x_1 : \ldots : x_n : \bot$$
$$\sqsubseteq x_0 : x_1 : \ldots : x_n : [\,]$$

This finite set of approximations is a chain. We have:

$$\bigsqcup \{\bot,\ x_0 : \bot,\ x_0 : x_1 : \bot,\ x_0 : x_1 : \ldots : x_n : \bot,$$
$$x_0 : x_1 : \ldots : x_n : [\,] \}$$
$$= x_0 : x_1 : \ldots : x_n : [\,]$$

Approximating Streams by Partial Lists

Similarly, streams can be approximated by partial lists, too:

$$\perp \sqsubseteq x_0 : \perp \sqsubseteq x_0 : x_1 : \perp \sqsubseteq x_0 : x_1 : \ldots : x_n : \perp$$
$$\sqsubseteq x_0 : x_1 : \ldots : x_n : x_{n+1} : \perp \sqsubseteq \ldots$$

This infinite set of approximations is a chain. We have:

$$\bigsqcup \{\perp,\ x_0 : \perp,\ x_0 : x_1 : \perp,\ x_0 : x_1 : x_2 : \perp, \ldots\} = xs$$

Computing Partial Approximations

The function approx gives approximations of any list, stream:

```
approx :: Integer -> [a] -> [a]
approx (n+1) []      = []
approx (n+1) (x:xs) = x : approx n xs
```

Note:

- n+1 matches only positive integers.
- Calling approx n xs with n smaller or equal to the length of xs will cause an error after generating the first n elements of the list, i.e., it generates the partial list

$$x_0 \; : \; x_1 \; : \ldots : \; x_{n-1} \; : \bot$$

- If n is greater than the length of xs, the call approx n xs generates the whole list xs.

Proof by Approximation

Lemma (6.4.6, Approximation)

For any list, stream xs holds:

$$\bigsqcup_{n=0}^{\infty} approx\ n\ xs \ = \ xs$$

Theorem (6.4.7, Approximation)

For any two lists, streams xs, ys hold:

$$xs = ys \ \Leftrightarrow \ \forall\, n \in I\!N.\ approx\ n\ xs \ = approx\ n\ ys$$

Proving Properties of Streams

Note:

- The Approximation Theorem 6.4.7 is an important means for proving properties of streams.
- The inductive proof principle for streams of Chapter 6.3.5 is justified by Theorem 6.4.7.

Chapter 6.4: Further Reading

- 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)

Chapter 6.5

Coinduction

Proof by Coinduction

...is another important principle

- for proving properties of infinite objects, e.g. equality of streams
- complements the principle of proof by approximation for proving properties of infinite objects (cf. Chapter 6.3.5)
- extends our tool box for proving properties of infinite objects like streams

Essence of Proof by Coinduction (1)

Proof by coinduction of equality of two infinite objects

- amounts to proving that the two objects exhibit the same observational behaviour.

For example, proving the equality of two streams *xs* and *ys* using the principle of proof by coinduction amounts to proving that

- *xs* and *ys* have the same heads
- the tails of *xs* and *ys* have the same observational behaviour

Essence of Proof by Coinduction (2)

Technically, proof by coinduction of the equality of two infinite objects *xs* and *ys* boils down to

- defining a bisimulation relation on *xs* and *ys*, and proving them to be bisimilar.

Formalizing this requires the notions of a labeled transition system and a bisimulation relation.

Labeled Transition Systems

Definition (6.5.1, Labeled Transition System)

A labeled transition system is a tripel (Q, A, T) consisting of

- a set of states Q
- a set of action labels A
- a ternary relation $T \subseteq Q \times A \times Q$, the transition relation.

Note:

- If $(q, a, p) \in T$, we write this as $q \xrightarrow{a} p$.

Bisimulations

Definition (6.5.2, (Greatest) Bisimulation)

Let (Q, A, T) be a labeled transition system.

A bisimulation on (Q, A, T) is a binary relation R on Q with the following properties.

If $q\,R\,p$ and $a \in A$ then

- If $q \xrightarrow{a} q'$ then there is a $p' \in Q$ with $p \xrightarrow{a} p'$ and $q'\,R\,p'$
- If $p \xrightarrow{a} p'$ then there is a $q' \in Q$ with $q \xrightarrow{a} q'$ and $q'\,R\,p'$

We denote the greatest bisimulation on Q by $\sim$.

Example

Consider the following decimal representations of $\frac{1}{7}$

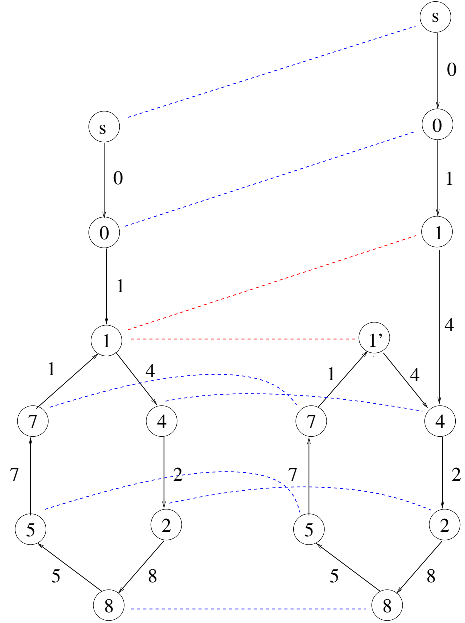
- $0.\overline{142857}$
- $0.1\overline{428571}$
- $0.14\overline{285714}$
- $0.142857142\overline{857142}$

and the relation R 'having the same infinite expansion' on decimal representations.

Then

- R is a bisimulation on decimal representations
- $0.\overline{142857}$, $0.1\overline{428571}$, $0.14\overline{285714}$, $0.142857142\overline{857142}$ are all bisimilar.

Illustration



Bisimilar

Definition (6.5.3, Bisimilar)

Let (Q, A, T) be a labeled transition system, and let $p, q \in Q$.

Then p and q are called bisimilar, if they are related by a bisimulation on Q.

Proof by Coinduction (1)

The general pattern of a proof by coinduction for proving the equality of infinite objects:

Let x and y be two infinite objects.

To prove that x and y are equal, show that they exhibit the same behaviour, i.e. prove that $x \sim y$:

$$a \sim b \Leftrightarrow \exists R. \, (R \text{ is a bisimulation, and } a \, R \, b)$$

Proof by Coinduction (2)

A proof matching the preceding pattern is called a

- proof by coinduction.

Next, we are going to show how to use this pattern to prove equality of streams.

Proof by Coinduction (3)

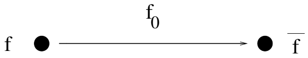
To this end, we introduce the following notation:

If $f = [f_0, f_1, f_3, f_4, f_5, \ldots]$ is a stream, then f_0 denotes the head and $\overline{f}$ the tail of f, i.e., $f = f_0 : \overline{f}$.

Note:

- A stream f can be considered a labeled transition system.

Illustration



Stream f as a labeled transition system

Equality of Streams

Let $f = [f_0, f_1, f_3, f_4, f_5, \ldots]$ and $g = [g_0, g_1, g_3, g_4, g_5, \ldots]$ be two streams.

Then

- f and g are equal iff they exhibit the same behaviour iff $\forall\, i \in \mathbb{N}_0.\ f_i = g_i$

This boils down to

- f and g are equal iff $f \sim g$, i.e., $f_0 = g_0$ and $\bar{f} \sim \bar{g}$ with $f \xrightarrow{f_0} \bar{f}$ and $g \xrightarrow{g_0} \bar{g}$.

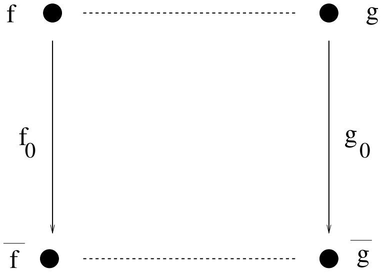
Stream Bisimulation

Definition (6.5.4, Stream Bisimulation)

A stream bisimulation on a set A is a relation R on $[A]$ with the following property.

If $f, g \in [A]$ and $f \; R \; g$ then both $f_0 = g_0$ and $\bar{f} \; R \; \bar{g}$.

Illustration



Bisimulation between two streams f and g

Proof by Coinduction w/ Stream Bisimulations

The general pattern of a proof by coinduction using stream bisimulations of $f \sim g$, where $f, g \in [A]$:

1. Define a relation R on $[A]$
2. Prove that R is a stream bisimulation, with $f\,R\,g$.

Chapter 6.5: Further Reading (1)

- 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.
- Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

Chapter 6.5: Further Reading (2)

- Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. Theoretical Computer Science 308:1-53, 2003.
- Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62, 1997.
- 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

...another important proof principle.

Fixed point induction allows proving properties of functions on ordered sets, such as complete partial orders, lattices, and the like (cf. Appendix).

Admissible Predicates

Definition (6.5.1, Admissible Predicate)

Let $(C, \sqsubseteq)$ be a complete partial order (CPO), and let $\psi : C \to I\!B$ be a predicate on C.

The predicate ψ is called admissible iff for every chain $D \subseteq C$ holds:

$$\text{if } \psi(d) = \mathit{true} \text{ for all } d \in D \text{ then } \psi(\bigsqcup D) = \mathit{true}$$

Fixed Point Induction

The general pattern of a proof by fixed point induction:

Theorem (6.5.2, Fixed Point Induction)

Let $(C, \sqsubseteq)$ be a complete partial order (CPO), let $f : C \to C$ be a continuous function on C, and let $\psi : C \to IB$ be an admissible predicate on C.

If for all $c \in C$ holds that

$$\psi(c) = true \;\; implies \;\; \psi(f(c))$$

then

$$\psi(\mu f) = true$$

where μf denotes the least fixed point of f.

Note

Streams

- form a domain resp. CPO (cf. Chapter 6.4 and Appendix)

Hence, fixed point induction is a relevant proof technique for a functional programmer.

Chapter 6.6: Further Reading

- 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

Other Approaches and Tools: A Selection (1)

- Programming by contracts (Vytiniotis et al., POPL 2013)
- Verifying equational properties of functional programs (Sonnex et al., TACAS 2012)
  - Tool Zeno: proof search based on induction and equality reasoning driven by syntactic heuristics.
- Verifying first-order and call-by-value recursive functional programs (Suter et al., SAS 2011)
  - Tool Leon: based on extending SMT with recursive programs.

Other Approaches and Tools: A Selection (2)

- Verifying higher-order functional programs (Unno et al., POPL 2013)
  - Tool MoCHi-X: prototype implementation of the type inference algorithm as an extension of the software model checker MoChi (Kobayashi et al, PLDI 2011).
- Verifying lazy Haskell (Mitchell et al., Haskell 2008)
  - Tool Catch: based on static analysis; can prove absence of pattern match failures; evaluated on 'real' programs.
- ...

Chapter 6: Further Reading (1)

- Martin Aigner, Günter M. Ziegler. *Proofs from the Book*. Springer-V., 4th edition, 2010.
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 5, Induction and Recursion)
- 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.

Chapter 6: Further Reading (2)

- 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)
- 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)

Chapter 6: Further Reading (3)

- 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.

Chapter 6: Further Reading (4)

- 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)
- 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 (5)

- 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.
- 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.
- 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)

Chapter 6: Further Reading (6)

- 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)
- 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?)

Chapter 6: Further Reading (7)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten)
- 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.
- 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.

Chapter 6: Further Reading (8)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 8, Reasoning about programs; Chapter 17.9, Proof revisited)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about programs; Chapter 17.9, Proof revisited)
- Simon Thompson. *Proof*. In *Research Directions in Parallel Functional Programming*, Kevin Hammond, Greg Michaelson (Eds.), Springer-V., Chapter 4, 93-119, 1999.

- 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.
- 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.

Part IV

Advanced Language Concepts

Chapter 7

Functional Arrays

Imperative Arrays

For imperative arrays holds:

- A value of the array can be accessed or updated in constant time.
- The update operation does not need extra space.
- There is no need for chaining the array elements with pointers as they can be stored in contiguous memory locations.

Lists and Functional Arrays

(Functional) lists

- do not enjoy the favorable list of characteristics of imperative arrays; most importantly, values of a list cannot be accessed or updated in constant time.
  - Using `(!!)` to access the *i*th element of a list takes a number of steps proportional to *i*.
- Lists can be arbitrarily long, potentially even infinite.

Functional arrays

- are designed and implemented to get as close as possible to the characteristics of imperative arrays.
  - Using `(!)` to access the *i*th element of an array takes a constant number of steps, regardless of *i*.
- Arrays are of a fixed size which must be defined at the time the array is (first) created.

Functional Arrays

Functional arrays

- are not part of the standard prelude `Prelude.hs` of Haskell.

Various libraries

- provide different kinds of functional arrays
  - `import Array`
  - `import Data.Array.IArray`
  - `import Data.Array.Diff`

Important variants of functional arrays

- Static arrays (w/out destructive update)
- Dynamic arrays (w/ destructive update)

Static Arrays

Creating static arrays

```
import Array
```

There are three functions for creating static arrays:

- `array` *bounds list_of_associations*
- `listArray` *bounds list_of_values*
- `accumArray` *f init bounds list_of_associations*

Creating Static Arrays

The three functions for creating static arrays in more detail:

- `array :: Ix a => (a,a) -> [(a,b)] -> Array a b`
  `array` *bounds list_of_associations*
- `listArray::(Ix a)=> (a,a) -> [b] -> Array a b`
  `listArray` *bounds list_of_values*
- `accumArray :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)] -> Array a b`
  `accumArray` *f init bounds list_of_associations*

The Type Class Ix

Ix denotes the class of types that are (mainly) used for indices of arrays.

- Members of the type class Ix must provide implementations of the functions
  - range
  - index
  - inRange
  - rangeSize
- Ix inherits from the type class Ord (and indirectly from the type class Eq):

```
class (Ord a) => Ix a where
 range     :: (a,a) -> [a]
 index     :: (a,a) -> a -> Int
 inRange   :: (a,a) -> a -> Bool
 rangeSize :: (a,a) -> Int
```

Creating Static Arrays: The 1st Mechanism

The first and most fundamental array creation mechanism:

- `array :: Ix a => (a,a) -> [(a,b)] -> Array a b`
  `array` *bounds list_of_associations*

Meaning of the arguments:

- *bounds*: gives the value of the lowest and the highest index in the array.

  Example: *bounds* of a
  - zero-origin vector of five elements: `(0,4)`
  - one-origin 10 by 10 matrix: `((1,1),(10,10))`

  Note: The values of the bounds can be arbitrary expressions.
- *list_of_associations*: a list of associations, where an association is of the form `(i,x)` meaning that the value of the array element `i` is `x`.

Examples

The expressions

```
a'  = 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]]
```

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)]
```

Properties of Array Creation

In general:

Arrays have type

- `Array a b` where
  - `a`: represents the type of the index
  - `b`: represents the type of the value

Moreover:

- An array is undefined if any specified index is out of bounds.
- If two associations in the association list have the same index, the value at that index is undefined.

  This means: `array` is strict in the bounds but non-strict (lazy) in the values. In particular, an array can thus contain 'undefined' elements.

Example

The computation of the Fibonacci numbers:

```
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 10 ->> array (1,10) [(1,0),(2,1),(3,1),
                          (4,2),(5,3),(6,5),
                          (7,8),(8,13),(9,21),
                          (10,34)]
```

Example (Cont'd)

More Applications:

```
fibs 5!5     ->> 3
fibs 10!10   ->> 34

fibs 100!10  ->> 34 -- Thanks to lazy evaluation
                    -- computation stops at
                    -- fibs 10!10

fibs 50!50   ->> 7.778.742.049
fibs 100!100 ->> 218.922.995.834.555.169.026

fibs 5!10 ->> Program error: Ix.index: index out
              of range
```

The Array Access Function (!)

The signature of the array access function (!):

```
(!) :: Ix a => Array a b -> a -> b
```

Recall: The index type must be an element of type class `Ix`, which defines operations specifically needed for index computations.

Example (Cont'd)

Note:

- The declaration of a in a where-clause is crucial for performance.
- The local declaration of a avoids creating new arrays during computation.

For comparison consider:

```
a n = array (1,n) ([(1,0), (2,1)] ++
                        [(i, a n!(i-1) + a n!(i-2))
                          | i <- [3..n]])

xfibs n = a n
```

Example (Cont'd)

Applications:

```
xfibs 3  ->> array (1,3) [(1,0),(2,1),(3,1)]
xfibs 5  ->> array (1,5) [(1,0),(2,1),(3,1),
                           (4,2),(5,3)]
xfibs 10 ->> array (1,10) [(1,0),(2,1),(3,1),(4,2),
                            (5,3),(6,5),(7,8),(8,13),
                            (9,21),(10,34)]

xfibs 5!5     ->> 3
xfibs 10!10   ->> 34
xfibs 25!20   ->> 4.181
xfibs 25!25   ->> ...takes too long to be feasible!
```

Note: Though correct, the evaluation of `xfibs n` is most inefficient due to the generation of new arrays during computation.

Creating Static Arrays: The 2nd Mechanism

The second array creation mechanism:

- `listArray :: (Ix a) => (a,a) -> [b] -> Array a b`
  `listArray` *bounds list_of_values*

Meaning of the arguments:

- *bounds*: gives the value of the lowest and the highest index in the array.
- *list_of_values*: a list of values.

The function `listArray`

- is useful for the frequenly occurring case where an array is constructed from a list of values in index order.

  Example:

  ```
  a'' = listArray (1,4) "face"
  ```

  ```
  a'' ->> array (1,4) [(1,'f'),(2,'a'),
                       (3,'c'),(4,'e')]
  ```

Creating Static Arrays: The 3rd Mechanism

The third array creation mechanism:

- `accumArray :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)] -> Array a b`
  `accumArray` *f init bounds list_of_associations*

...removes the restriction that a given index may appear at most once in the association list. Instead, 'conflicting' indices are accumulated via a function *f*.

Meaning of the arguments:

- *f*: an accumulation function.
- *init*: gives the (default) value the entries of the array shall be initialized with.
- *bounds*: gives the value of the lowest and the highest index in the array.
- *list_of_associations*: a list of associations.

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
```

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
```

A Prime Number Test (Cont'd)

More Applications:

```
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
```

Array Operators

Array operators are:

- `!`: array subscripting.
- `bounds`: yields bounds of an array.
- `indices`: yields list of indices of an array.
- `elems`: yields list of elements of an array.
- `assocs`: yields list of associations of an array.
- `//`: array updating – the operator `//` takes an array and a list of associations and returns a new array identical to the left argument except for every element specified by the right argument list.

  This means: `//` does not perform a destructive update!
- ...

Array Operators (Cont'd)

- `(!)  :: (Ix a) => Array a b -> a -> b`
- `bounds :: (Ix a) => Array a b (a,a)`
- `indices :: (Ix a) => Array a b -> [a]`
- `elems :: (Ix a) => Array a b -> [b]`
- `assocs :: (Ix a) => Array a b -> [(a,b)]`
- `(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b`
- ...

Illustrating the Usage of Array Operators

Let

```
m = array ((1,1), (2,3)) [((i,j), (i*j))
                           | i<-[1..2], j<-[1..3]]
```

Then

```
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

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)]
```

Illustrating the Update Operator

The histogram function:

```
histogram (lower,upper) xs
 = updHist (array (lower,upper)
                  [(i,0) | i<-[lower..upper]])
            xs

updHist a [] = a
updHist a (x:xs) = updHist (a // [(x, (a!x + 1))]) xs
```

Application:

```
histogram (0,9) [3,1,4,1,5,9,2]
 ->> array (0,9) [(0,0),(1,2),(2,1),(3,1),(4,1),
                  (5,1),(6,0),(7,0),(8,0),(9,1)]
```

Illustrating the accum Operator

Instead of replacing the old value, values with the same index could also be combined using the predefined:

- `accum :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)] -> Array a b`
  `accum` *function array list_of_associations*

Application:

```
accum (+) m [((1,1),4), ((2,2),8)]
 ->> 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 is a new matrix identical to `m` except for the elements `(1,1)` and `(2,2)` to which `4` and `8` have been added, respectively.

Higher-Order Array Functions

Higher-order functions can be defined on arrays just as on lists.

For illustration consider:

- The expression

  ```
  map (\x -> x*10) a
  ```

  ...creates a new array where all elements of a are multiplied by 10.

- The expression

  ```
  ixmap b f a = array b [(k, a ! f k) | k<-range b]
  ```

  ...
  with

  ```
  ixmap :: (Ix a, Ix b) => (a,a) -> (a -> b)
                                -> Array b c -> Array a c
  ```

Higher-Order Array Functions (Cont'd)

The functions `row` and `col` return a row and a column of a matrix:

```
row :: (Ix a, Ix b) =>
              a -> Array (a,b) c -> Array b c
row i m = ixmap (l', u') (\j->(i,j)) m
    where ((l,l'),(u,u')) = bounds m

col :: (Ix a, Ix b) =>
              a -> Array (b,a) c -> Array b c
col j m = ixmap (l,u) (\i->(i,j)) m
    where ((l,l'),(u,u')) = bounds m
```

Higher-Order Array Functions (Cont'd)

Applications:

```
row 1 m ->> array (1,3) [(1,1),(2,2),(3,3)]
row 2 m ->> array (1,3) [(1,2),(2,4),(3,6)]
row 3 m ->> array (1,3) [(1,
            Program error: Ix.index: index out of
            range

col 1 m ->> array (1,2) [(1,1),(2,2)]
col 2 m ->> array (1,2) [(1,2),(2,4)]
col 3 m ->> array (1,2) [(1,3),(2,6)]
col 4 m ->> array (1,2) [(1,
            Program error: Ix.index: index out of
            range
```

Dynamic Arrays

Creating dynamic arrays

```
import Data.Array.Diff
```

Instead of

- type `Array`

we now have to use

- type `DiffArray`

...everything else remains the same.

Summing up

Static Arrays

- Access operator (!): access to each array element in constant time.
- Update operator (//): no destructive updates; instead an identical copy of the argument array is created except of those elements which were 'updated.' Updates thus do not take constant time.

Dynamic Arrays

- Update operator (//): destructive updates; updates take constant time per index.
- Access operator (!): access to array elements may sometimes take longer as for static arrays.

Summing up (Cont'd)

Recommendation

- Dynamic arrays should only be used if constant time updates are crucial for the application.
- Often, updates can completely be avoided by smartly written recursive array constructions (cp. the prime number test in this chapter).

Chapter 7: Further Reading (1)

- 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

Concrete vs. Abstract Data Types (1)

Concrete Data Types (CDTs)

- A new CDT is specified by naming its values.
- With the exception of functions, each value is of a type is described by a unique expression in terms of constructors.
- Using definition by pattern matching as a basis, these expressions can be generated, inspected, and modified in various ways.
- There is no need to specify the operations associated with a type.
- The Haskell means for defining CDTs are algebraic data type definitions.

Concrete vs. Abstract Data Types (2)

Abstract Data Types (ADTs)

- A new ADT is not specified by naming its values but by naming its operations.
- How values are represented is thus less important than what operations are provided for manipulating them, whose meaning, of course, has to be described
  - degree of freedom for the implementation!
  - Information hiding!
- There is no dedicated means in Haskell for defining ADTs; ADTs, however, can be defined using modules.

Concrete vs. Abstract Data Types (3)

Implementing an ADT

- When implementing an ADT, a representation of its values has to be provided, and a definition of the operations of the type in terms of this representation.
- The representation can be chosen e.g. for grounds of simplicity or efficiency.
- It has to be shown that the implemented operations satisfy the prescribed relationships.

In the following

...we consider abstract data types for

- Stacks
- Queues
- Priority Queues
- Tables

Chapter 8.1

Stacks

The Abstract Data Type Stack (1)

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

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

push       :: a -> Stack a -> Stack a
pop        :: Stack a -> Stack a
top        :: Stack a -> a
emptyStack :: Stack a
stackEmpty :: Stack a -> Bool
```

Note: In a stack elements are removed in a last-in/first-out (LIFO) order.

The Abstract Data Type Stack (2)

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

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

push x s = Stk x s

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

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

emptyStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _        = False
```

The Abstract Data Type Stack (3)

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

```
newtype Stack a = Stk [a]

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

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

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

emptyStack = Stk []

stackEmpty (Stk []) = True
stackEmpty (Stk _)  = False
```

Displaying Stacks (1)

Note:

- The constructors `EmptyStk` and `Stk` are not exported from the module.
- This implies that a user of the module can not use or create a `Stack` by any other way than the operations exported by the module
- While this is actually so desired, the user can also not display a value of type `Stack` except for the crude and cumbersome way of completely popping the whole stack.

Next, we describe and compare to ways to display stacks and their elements more elegantly.

Displaying Stacks (2)

The easy way: Using a `deriving`-clause

```
data Stack a = EmptyStk
             | Stk a (Stack a) deriving Show

newtype Stack a = Stk [a] deriving Show
```

Effect:

```
push 3 (push 2 (push 1 emptyStack))
  ->> Stk 3 (Stk 2 (Stk 1 EmptyStk))

push 3 (push 2 (push 1 emptyStack))
  ->> Stk [3,2,1]
```

Displaying Stacks (3)

Using the `deriving`-clause for type class `Show`:

Advantage

- Simplicity, no effort.

Disadvantage

- The implementation of the ADT `Stack` is disclosed to the programmer (though the user cannot access the representation in any way outside the module definition of the ADT `Stack`).

Displaying Stacks (4)

A smarter solution:

```
instance (Show a) => Show (Stack a) where
 showsPrec _ EmptyStk str = showChar '-' str
 showsPrec _ (Stk x s) str
   = shows x (showChar '|' (shows s str))

instance (Show a) => Show (Stack a) where
 showsPrec _ (Stk []) str = showChar '-' str
 showsPrec _ (Stk (x:xs)) str
   = shows x (showChar '|' (shows (Stk xs) str))
```

Effect:

```
push 3 (push 2 (push 1 emptyStack)) ->> 3|2|1|-
```

Displaying Stacks (5)

This way:

- The implementation of the ADT `Stack` remains hidden. It is not disclosed to the user.
- The output is the same for both implementations!

Note:

- The first argument of `showsPrec` is an unused precedence value.

Last but not least

An implementation of stacks in terms of

- predefined lists in Haskell: `type Stack a = [a]`

would be possible, too.

Advantage

- Even less conceptual overhead as for the implementation in terms of `newtype Stack a = Stk [a]`

Disadvantage

- All predefined functions on lists would be available on stacks, too.
- Many of these, however, e.g. for reversing a list, for picking some arbitrary element, are not meaningful for stacks.
- Implementing stacks in terms of predefined lists would not automatically exclude the application of such meaningless functions but require to explicitly abstain from them. Conceptually, this is disadvantageous.

Chapter 8.2

Queues

The Abstract Data Type Queue (1)

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

```
module Queue (Queue,emptyQueue,queueEmpty,
              enQueue,deQueue,front) where

emptyQueue :: Queue a
queueEmpty :: Queue a -> Bool
enQueue    :: a -> Queue a -> Queue a
deQueue    :: Queue a -> Queue a
front      :: Queue a -> a
```

Note: In a queue elements are removed in a first-in/first-out (FIFO) order.

The Abstract Data Type Queue (2)

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

```
newtype Queue a = Q [a]

emptyQueue = Q []

queueEmpty (Q []) = True
queueEmpty _      = False

enQueue x (Q q) = Q (q ++ [x])

deQueue (Q [])     = error "deQueue: empty queue"
deQueue (Q (_:xs)) = Q xs

front (Q [])    = error "front: empty queue"
front (Q (x:_)) = x
```

Displaying Queues

The easy way: Using a `deriving`-clause

```
newtype Queue a = Q [a] deriving Show
```

Advantages, disadavantages:

- Cp. Chapter 8.1.

Chapter 8.3

Priority Queues

The Abstract Data Type PQueue (1)

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

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

emptyPQ :: PQueue a
pqEmpty :: PQueue a -> Bool
enPQ    :: (Ord a) => a -> PQueue a -> PQueue a
dePQ    :: (Ord a) => PQueue a -> PQueue a
frontPQ :: (Ord a) => PQueue a -> a
```

Note: In a priority queue each entry has a priority associated with it. The dequeue operation always removes the entry with the highest (or lowest) priority. Technically, this is ensured by the enqueue operation, which places a new element according to its priority in a queue.

The Abstract Data Type PQueue (2)

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

```
newtype PQueue a = PQ [a]

emptyPQ = PQ []

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

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

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

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

Displaying Priority Queues

The easy way: Using a `deriving`-clause

```
newtype PQueue a = PQ [a] deriving Show
```

Advantages, disadavantages:

- Cp. Chapter 8.1.

Chapter 8.4

Tables

The Abstract Data Type Table (1)

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

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

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

Note:

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

The Abstract Data Type Table (2)

A user-invisible implementation of Table as a function:

```
newtype Table a b = Tbl (b -> a)

newTable assocs =
 foldr updTable
       (Tbl (\_ -> eror "updTable: item not found"))
       assocs

findTable (Tbl f) i = f i

updTable (i,x) (Tbl f) = Tbl g
 where g j | j==i       = x
           | otherwise = f j
```

Displaying Tables Represented as Functions

Using an `instance`-clause

```
instance Show (Table a b) where
  showsPrec _ _ str = showString "<<A Table>>" str
```

The Abstract Data Type Table (3)

A user-invisible implementation of Table as a list:

```
newtype Table a b = Tbl [(b,a)]

newTable t = Tbl t

findTable (Tbl []) i
 = error "findTable: item not found"
findTable (Tbl ((j,v):r)) i
 | i==j      = v
 | otherwise = findTable (Tbl r) i

updTable e (Tbl []) = Tbl [e]
updTable e'@(i,_) (Tbl (e@(j,_):r))
 | i==j       = Tbl (e':r)
 | otherwise  = Tabl (e:r')
 where Tbl r' = updTable e' (Tbl r)
```

Displaying Tables Represented as Lists

The easy way: Using a `deriving`-clause

```
newtype Table a b = Tbl [(b,a)] deriving Show
```

Advantages, disadavantages:

- Cp. Chapter 8.1.

The Abstract Data Type Table (4)

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

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

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

Note:

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

The Abstract Data Type Table (5)

A user-invisible implementation of Table as an Array:

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

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

findTable (Tbl a) i = a ! i

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

The Abstract Data Type Table (6)

Note:

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

Displaying Tables Represented as Arrays

The easy way: Using a `deriving`-clause

```
newtype Table a b = Tbl (Array b a) deriving Show
```

Advantages, disadavantages:

- Cp. Chapter 8.1.

Chapter 8.5

Summing Up

Summing up

Benefits of using abstract data types:

- Information hiding: Only the interface is publicly known; the implementation itself is hidden. This offers:
  - Security of the data (structure) from uncontrolled or unintended/not admitted access.
  - Simple exchangeability of the underlying implementation (e.g. simplicity vs. performance).
  - Work-sharing of implementation.

There are many more implementations of data types in terms of an abstract data type. E.g.:

- Sets
- Heaps
- (Binary Search) Trees
- Arrays
- ...

Arrays: An Abstract Data Type

```
module Array (
       module Ix, -- export all of Ix for convenience
       Array, array, listarray (!), bounds, indices,
       elems, assocs, accumArray, (//),
       accum, ixmap ) where

import Ix
infixl 9 !, //
data (Ix a) => Array a b = ... -- Abstract

array       :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray   :: (Ix a) => (a,a) -> [b] -> Array a b
(!)         :: (Ix a) => Array a b -> a -> b
bounds      :: (Ix a) => Array a b (a,a)
indices     :: (Ix a) => Array a b -> [a]
elems       :: (Ix a) => Array a b -> [b]
assocs      :: (Ix a) => Array a b -> [(a,b)]
```

Arrays: An Abstract Data Type (Cont'd)

```
accumArray :: (Ix a) => (b -> c -> b) -> b
                        -> (a,a) -> [(a,c)] -> Array a b
(//)       :: (Ix a) => Array a b -> [(a,b)]
                                     -> Array a b
accum      :: (Ix a) => (b -> c -> b) -> Array a b
                                 -> [(a,c)] -> Array a b
ixmap      :: (Ix a, Ix b) => (a,a) -> (a -> b)
                               -> Array b c -> Array a c

instance Functor (Array a) where...
instance (Ix a, Eq b) => Eq (Array a b) where...
instance (Ix a, Ord b) => Ord (Array a b) where...
instance (Ix a, Show a, Show b)
                => Show (Array a b) where...
instance (Ix a, Read a, Read b)
                => Read (Array a b) where...
```

Arrays: An Abstract Data Type (Cont'd)

See also:

- Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 173-178, 2003.

Chapter 8: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Chapter 10, Arrays, Listen und Stacks)
- 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)
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.5, Abstract Types and Modules)

Chapter 8: Further Reading (2)

- 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.
- 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)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract Data Types)

Chapter 8: Further Reading (4)

- 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

The Type Class Monoid

Monoids are instances of the type class Monoid:

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

...where the implementations of the monoid operations need to satisfy the so-called monoid laws.

Intuitively:

- A monoid is made up of an associative binary function `mappend`, and an element `mempty` that acts as an identity for with respect to the function `mappend`.
- The function `mconcat` takes a list of monoid values and reduces them to a single monoid value by using `mappend`.

The Laws of Monoid

Members of the type class Monoid must satisfy the following three laws:

```
mempty `mappend` x = x                            (MoL1)
x `mappend` mempty = x                            (MoL2)
(x `mappend` y) `mappend` z =
    x `mappend` (y `mappend` z)                   (MoL3)
```

Intuitively:

- The first two laws (MoL1) and (MoL2) require that mempty is the identity with respect to mappend.
- The third law (MoL3) requires that function mappend is associative.

Note:

- It needs to be proven that these laws are satisfied by a concrete instance of class Monoid. This is a proof obligation for the programmer!

Remarks

Note:

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

Examples: The List Monoid (1)

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

Examples:

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

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

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

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

Examples: The List Monoid (2)

More Examples:

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

mempty :: [a] ->> []
```

Note:

The monoid laws do not require commutativity of the binary operation `mappend`:

```
"Semester " ‘mappend‘ "Holiday"
      ->> "Semester Holiday"
```

but

```
"Holiday " ‘mappend‘ "Semester"
      ->> "Holiday Semester"
```

Examples: The Product and Sum Monoids (1)

```
newtype Product a = Product { getProduct :: a }
  deriving (Eq, Ord, Read, Show, Bounded)

newtype Sum a = Sum { getSum :: a }
  deriving (Eq, Ord, Read, Show, Bounded)


instance Num a => Monoid (Product a) where
  mempty  = Product 1
  Product x `mappend` Product y = Product (x*y)

instance Num a => Monoid (Sum a) where
  mempty  = Sum 0
  Sum x `mappend` Sum y = Product (x+y)
```

Examples: The Product and Sum Monoids (2)

Examples:

```
getProduct $ Product 3 `mappend` Product 7 ->> 21
getSum $ Sum 17 `mappend` Sum 4 ->> 21

getProduct $ Product 3 `mappend` Product 7
    `mappend` Product 11 ->> 231
getSum $ Sum 3 `mappend` Sum 7 `mappend` Sum 11
                          ->> 21

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
```

Examples: The Any and All Monoids (1)

```
newtype Any = Any { getAny :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)

newtype All = All { getAll :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)


instance Monoid Any where
  mempty  = Any False
  Any x `mappend` Any y = Any (x || y)
    -- Any because True if any argument is true.

instance Monoid All where
  mempty  = All True
  All x `mappend` All y = All (x && y)
    -- All because True if all arguments are true.
```

Examples: The Any and All Monoids (2)

Examples:

```
getAny $ Any True `mappend` Any False ->> True
getAll $ All True `mappend` All False ->> False

getAny $ mempty `mappend` Any False ->> False
getAll $ All True `mappend` mempty ->> True

getAny . mconcat . map Any $ [False,True,False,False]
   ->> True
getAll . mconcat . map All $ [False,True,True,False]
   ->> False
```

Remark

Note:

- For the Product, Sum, Any, and All monoids the binary function `mappend` happens to be commutative, too.

Examples: The Ordering Monoid (1)

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

Note:

- The definition of the binary function mappend leads to 'alphabetically' comparing lists of arguments.
- For the Ordering monoid the binary function mappend fails to be commutative:

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

Examples: The Ordering Monoid (2)

Example:

The declaration

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y
  = let a = length x `compare` length y
                              -- fst priority
        b = x `compare` y -- snd priority
    in if a == EQ then b else a
```

can equivalently be rewritten as

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

by using the monoid properties.

Examples: The Ordering Monoid (3)

As expected we get

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

but

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

Examples: The Ordering Monoid (4)

Comparison criteria can easily be added and prioritisized.

E.g., the below extension of `lengthCompare` takes the number of vowels as the second most important comparison criteron:

```
lengthCompareExt :: String -> String -> Ordering
lengthCompareExt x y
   = (length x `compare` length y) -- fst priority
     `mappend` (vowels x `compare` vowels y)
                                    -- snd priority
     `mappend` (x `compare` y)      -- thd priority
where vowels = length . filter (`elem` "aeiou")
```

As expected we get:

```
lengthCompareExt "his" "ancestors" ->> LT
lengthCompare "his" "ancestors"    ->> LT
lengthCompare "his" "ant"          ->> GT
```

Summing up (1)

Monoids are especially useful for defining

- folds over various data structures.

While this seems obvious for

- lists

it also holds for many other data structures including

- trees

and many others.

Summing up (2)

This has led to the introduction of the type class Foldable (see module Data.Foldable):

```
class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b
  foldl :: (a -> b -> a) -> a -> f b -> a
  ...
```

whose fold operations generalize those on lists to foldable types, i.e., instances of the class Foldable:

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

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

Outline

In Chapter 7 of LVA 185.A03 we were going

- from functions to higher-order functions

In this chapter we are going

- from type classes to higher-order type classes

Chapter 9.1

Motivation

Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (1)

(siehe Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach, Addison-Wesley, 1999, S. 7f.)

Betrachte folgende Beispiele:

- Fakultätsfunktion:

```
fac n | n==0   = 1
      | n>0    = n * fac (n-1)
```

- Summe der *n* ersten natürlichen Zahlen:

```
natSum n | n==0  = 0
         | n>0   = n + natSum (n-1)
```

- Summe der *n* ersten natürlichen Quadratzahlen:

```
natSquSum n | n==0  = 0
            | n>0   = n*n + natSquSum (n-1)
```

Recall "Kapitel 7, LVA 185.A03"

Funktionale Abstraktion höherer Stufe (2)

Beobachtung:

- Die Definitionen von `fac`, `sumNat` und `sumSquNat` folgen demselben Rekursionsschema.

Dieses zugrundeliegende gemeinsame Rekursionsschema ist gekennzeichnet durch:

- Festlegung eines Wertes der Funktion im
  - Basisfall
  - verbleibenden rekursiven Fall als Kombination des Argumentwerts `n` und des Funktionswerts für `n-1`

Recall “Kapitel 7, LVA 185.A03”

Funktionale Abstraktion höherer Stufe (3)

Dies legt nahe:

- Obiges Rekursionsschema, gekennzeichnet durch Basisfall und Funktion zur Kombination von Werten, herauszuziehen (zu abstrahieren) und musterhaft zu realisieren.

Wir erhalten:

- Realisierung des Rekursionsschemas

```
recScheme base comb n
  | n==0  = base
  | n>0   = comb n (recScheme base comb (n-1))
```

Recall "Kapitel 7, LVA 185.A03"

Funktionale Abstraktion höherer Stufe (4)

Funktionale Abstraktion höherer Stufe:

```
fac n       = recScheme 1 (*) n

natSum n    = recScheme 0 (+) n

natSquSum n = recScheme 0 (\x y -> x*x + y) n
```

Noch einfacher: In argumentfreier Ausführung

```
fac       = recScheme 1 (*)

natSum    = recScheme 0 (+)

natSquSum = recScheme 0 (\x y -> x*x + y)
```

Recall "Kapitel 7, LVA 185.A03"

Funktionale Abstraktion höherer Stufe (5)

Unmittelbarer Vorteil obigen Vorgehens:

- Wiederverwendung und dadurch
  - kürzerer, verlässlicherer, wartungsfreundlicherer Code

Erforderlich für erfolgreiches Gelingen:

- Funktionen höherer Ordnung; kürzer: Funktionale.

  Intuition: Funktionale sind (spezielle) Funktionen, die Funktionen als Argumente erwarten und/oder als Resultat zurückgeben.

Recall "Kapitel 7, LVA 185.A03"

Funktionale Abstraktion höherer Stufe (6)

Illustriert am obigen Beispiel:

- Die Untersuchung des Typs von `recScheme`

  ```
  recScheme :: Int -> (Int -> Int -> Int) -> Int
  ```

  zeigt:
  - `recScheme` ist ein Funktional!

In der Anwendungssituation des Beispiels gilt weiter:

	Wert i. Basisf. (base)	Fkt. z. Kb. v. W. (comb)
`fac`	1	`(*)`
`natSum`	0	`(+)`
`natSquSum`	0	`\x y -> x*x + y`

Let's switch to a slightly more complex example

The higher-order function map on

- Lists

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

- (Binary) Trees

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

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

From Higher-Order Functions

...to Higher-Order Type Classes.

It is worth noting that the implementations of

- `mapList`
- `mapTree`

like the implementations of `fac`, `natSum`, and `natSquSum` are structurally similar, too.

This similarity suggests

- striving for a function genericMap that covers `mapList`, `mapTree`, and more

...and leads us to the

- (type) constructor class `Functor`.

Chapter 10.2

Constructor Class Functor

The Constructor Class Functor

Functors are instances of the constructor class Functor:

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

...where the implementation of the functor operation fmap needs to satisfy the so-called functor laws.

Note:

- The argument f of Functor is applied to type variables. This means, f is not a type variable but a type constructor that is applied to the type variables a and b.
- Members of (type) constructor classes are type constructors, no types.
- The functor operation of an instance of Functor takes a polymorphic function g :: a -> b and yields a polymorphic function g' :: f a -> f b, e.g., g :: Int -> String, and g' :: Month Int -> Season String.

The Type Class Eq

For comparison recall the type class Eq:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y      = not (x==y)
  x == y      = not (x/=y)
```

Note:

- The argument a of Eq is a type variable. Functions declared in Eq operate on a; a itself operates on nothing.
- This holds as well for the other type classes we considered so far such as Ord, Num, Fractional, etc.

Constructor Classes vs. Type Classes

In principle, these are similar concepts but with different members.

- Constructor classes (`Functor, Monad,...`)
  - have type constructors (e.g., `Tree,[],(,),...`) as members.
- Type classes (`Eq a, Ord a, Num a,...`)
  - have types (e.g., `Tree a,[a],(a,a),...`) as members.

Type constructors are

- functions, which from given types construct new ones.

  Examples: Tuple constructors `(,), (,,), (,,,)`; List constructor `[]`; Functional constructor `->`; Input/output constructor `IO,...`

The Laws of Functor

Members of the constructor class Functor must satisfy the following two laws:

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

Intuitively:

- The "shape of the container type" is preserved.
- The contents of the container is not regrouped.

Note:

- It needs to be proven that these two laws are satisfied by a concrete instance of class Functor such as trees, lists, etc. This is a proof obligation for the programmer!

Lists and Trees as Instances of `Functor` (1)

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

instance Functor Tree where
  fmap g (Leaf v) = Leaf (g v)
  fmap g (Node v l r)
    = Node (g v) (fmap g l) (fmap g r)
```

Note:

- The symbol [ ] is used above in two roles, as a
  - type constructor in the line instance Functor [a] where...
  - value of some list type in the line fmap g [] = [].
- The declarations instance Functor [a] where... and instance Functor (Tree a) where... were incorrect, since [a] and (Tree a) are types, no type constructors.

Lists and Trees as Instances of Functor (2)

The next instance declarations are equivalent but more concise:

```
instance Functor [] where
  fmap = mapList    -- user-defined mapList

instance Functor [] where
  fmap = map        -- predefined map

instance Functor Tree where
  fmap = mapTree    -- user-defined mapTree
```

Lists and Trees as Instances of `Functor` (3)

Applications:

```
t = Node 2 (Node 3 (Leaf 5) (Leaf 7)) (Leaf 11)

fmap (*2) t
 ->> Node 4 (Node 6 (Leaf 10) (Leaf 14)) (Leaf 22)
fmap (^3) t
 ->> Node 8 (Node 27 (Leaf 125) (Leaf 343))
            (Leaf 1331)

fmap (*2) [1..5] ->> [2,4,6,8,10]
fmap (^3) [1..5] ->> [1,8,27,64,125]
```

Quintessence

The function `fmap` of constructor class `Functor` is

- the function `genericMap`

that we were looking and striving for.

Members of the constructor class `Functor` can be both

- pre-defined and user-defined type constructors.

Predefined Type Constructors

Examples of predefined type constructors:

- `( , )`, `( , , )`, `( , , , )`, etc.: constructors for tuple types
- `[ ]`: constructor for list types
- `(->)`: constructor for functional types

Notational Remarks

The following notations are equivalent:

- `(a,b)` is equivalent to `(,) a b`
  `(a,b,c)` is eqivalent to `(,,) a b c`, etc.
- `[a]` is equivalent to `[] a`
- `f -> g` is equivalent to `(->) f g`
- `T a b` is equivalent to `((T a) b)` (i.e., associativity to the left as for function application)

Illustration (1)

The signatures of the functions `fac` and `list2pair`

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

list2pair :: [a] -> (a,a)
list2pair (x : (y : _ )) = (x,y)
list2pair (x : _)        = (x,x)
```

Illustration (2)

...can equivalently be specified as follows:

```
fac :: (->) Int Int

list2pair :: [] a -> (a,a)
list2pair :: [a] -> (,) a a
list2pair :: (->) [a] (a,a)
list2pair :: [] a -> (,) a a
...
list2pair :: (->) ([] a) ((,) a a)
```

Nonetheless, more easily understandable (maybe only because we are more accustomed to) seem the "classical" variants:

```
fac :: Int -> Int

list2pair :: [a] -> (a,a)
```

More Examples: Maybe as Functor

```
data Maybe a = Nothing | Just a

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

Example:

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

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

More Examples: `IO` as Functor (1)

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

Example:

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

is equivalent to

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

More Examples: IO as Functor (2)

```
import Data.Char
import Data.List

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

has the effect of

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

Applied to

```
hello there
```

we get

```
E-R-E-H-T- -O-L-L-E-H
```

More Examples: Either as Functor (1)

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

Either has two type parameters. Hence, only (Either a) can be made an instance of Functor:

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

Example:

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

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

More Examples: Either as Functor (2)

Note that

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

would not be meaningful. Think about why not. Think about what this would mean for the types replaced for a and b.

An Antiexample (1)

Consider the type CounterMaybe

```
data CounterMaybe a = CNothing
                    | CJust Int a deriving (Show)
```

and make it an instance of class Functor:

```
instance Functor CounterMaybe where
 fmap f CNothing = CNothing
 fmap f (CJust counter x) = CJust (counter+1) (f x)
```

An Antiexample (2)

We get:

```
CNothing ->> CNothing
CJust 0 "haha" ->> Cjust 0 "haha"
CNothing :: CMaybe a
CJust 0 "haha" :: CMaybe [Char]
CJust 100 [1,2,3] ->> CJust 100 [1,2,3]
```

We also get:

```
fmap (++ "ha") (CJust 0 "ho")
  ->> CJust 1 "hoha"
fmap (++ "he") (fmap (++ "ha") (CJust 0 "ho")
  ->> CJust 2 "hohahe"
fmap (++ "blah") CNothing
  ->> CNothing
```

An Antiexample (3)

However, we get

```
fmap id (CJust 0 "haha")
  ->> CJust 1 "haha"
```

whereas

```
id (CJust 0 "haha")
  ->> CJust 0 "haha"
```

- This shows that `fmap` defined for `CounterMaybe` violates the first `Functor` law: `fmap id = id`
- `CounterMaybe` can not be considered a valid instance of class `Functor`.

Summing up (1)

The constructor class `Functor`:

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

Laws of `Functor`:

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

Summing up (2)

Some instance declarations:

```
instance Functor Tree where
  fmap g (Leaf v) = Leaf (g v)
  fmap g (Node v l r)
    = Node (g v) (fmap g l) (fmap g r)

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

Summing up (3)

More concise instance declarations:

```
instance Functor [] where
  fmap = mapList    -- user-defined mapList

instance Functor [] where
  fmap = map        -- predefined map

instance Functor Tree where
  fmap = mapTree    -- user-defined mapTree
```

Summing up (4)

Some applications of the `Functor` function `fmap`:

```
t = 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)

fmap (*2) [1..5] ->> [2,4,6,8,10]
fmap (^3) [1..5] ->> [1,8,27,64,125]
fmap (3^) [1..5] ->> [3,9,27,81,243]
```

Chapter 10.3

Applicative Functors

Motivation

Comparing

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

and

```
(->) r l
```

suggests that

```
((->) r)
```

can be made an instance of class `Functor` just as

```
(Either a)
```

can be made.

⇝ This leads us to applicative functors, i.e., to functions as functors.

Functions as Functors (1)

```
instance Functor ((->) r) where
  fmap f g = (\x -> f (g x))
```

The type of `fmap`

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

for this instance of `Functor` becomes

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

Using infix notation for `->` this becomes:

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

Functions as Functors (2)

In effect, this means:

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

stands for function composition!

Hence, the instance definition can more concisely be given by:

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

Functions as Functors (3)

Examples:

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


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

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

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

Note:

- Calling `fmap` as an infix operation emphasizes the similarity of `fmap` and function composition `.`

fmap and Currying (1)

Reconsidering

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

we get:

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

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

This can be considered

- lifting of an `a -> b` function to an `f a -> f b` function.

fmap and Currying (2)

This shows that fmap can be thought of in two ways:

- Uncurried: As a function that takes a function and a function value and then maps that function over the functor value.
- Curried: As a function that takes a function and lifts that function so it operates on functor values.

fmap and Currying (3)

Examples:

```
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 "blah")
 ->> Right ["blah","blah","blah"]

fmap (replicate 3) Nothing
 ->> Nothing

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

Towards Applicative Functors

Some Examples:

```
fmap (++) (Just "hey") :: 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]

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

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

The Type Constructor Class Applicative

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

Maybe as Applicative

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

Examples:

```
Just (+3) <*> Just 9  ->> Just 12
Just (+3) <*> Just 10 ->> Just 13

Just (++ "hello") <*> Nothing ->> Nothing
Nothing <*> Just "hello"      ->> Nothing
```

Examples:

```
pure (+) <*> Just 3 <*> Just 5  ->> 8
pure (+) <*> Just 3 <*> Nothing ->> Nothing
pure (+) <*> Nothing <*> Just 5 ->> Nothing
```

The operator (<*>) is left-associative:

```
pure (+) <*> Just 3 <*> Just 5 =
             (pure (+) <*> Just 3) <*> Just 5
```

Defining an Infix Alias for fmap (1)

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

Note:

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

would be valid as well:

- Type variables (like the `f` in the function declaration) are independent of parameter names (like the `f` in the function body) and other value names.

Defining an Infix Alias for fmap (2)

Examples:

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

Lists [] as Applicative (1)

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

Examples:

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

Lists [] as Applicative (2)

More Examples:

```
[(*0),(+100),(^2)] <*> [1,2,3]
  ->> [0,0,0,101,102,103,1,4,9]
[(+),(*)] <*> [1,2] <*> [3,4]
  ->> [4,5,5,6,3,4,6,8]
(++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
  ->> ["ha?","ha!","ha.","heh?","heh!","heh.",
       "hmm?","hmm!","hmm."]
```

Also list comprehension can be replaced this way:

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

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

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

IO as Applicative (1)

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

More Examples:

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

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

IO as Applicative (2)

```
main = do
 a <- (++) <$> getLine <*> getLine
 putStrLn $
   "The concatenation of the two lines is: " ++ a
```

Functions (->) r as Applicative (1)

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

Examples:

```
(pure 3) "Hello" ->> 3
pure 3 "Hello"   ->> 3  (left-associativity)

(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
(+) <$> (+3) <*> (*100) $ 5 ->> 508

(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
  ->> [8.0,10.0,2.5]
```

Zip Lists as Applicative (1)

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs =
    ZipList (zipWith (\f x -> f x) fs xs)
```

Zip Lists as Applicative (2)

Examples:

```
getZipList $
  (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
  ->> [101,102,103]

getZipList $
  (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
  ->> [101,102,103]

getZipList $
  max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
  ->> [5,3,3,4]

getZipList $
  (,,) <$> ZipList "dog" <*> ZipList "cat"
                                      <*> ZipList "rat"
  ->> [('d','c','r'),('o','a','a'),('g','t','t')]
```

The Laws of Applicative

Members of the constructor class Applicative must satisfy the following laws:

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

Useful Functions for Applicative (1)

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

Examples:

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

Useful Functions for Applicative (2)

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

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

Chapter 10.4

Kinds of Types and Type Constructors

Kinds of Types and Type Constructors

Like values

- types and
- type constructors

have types, too.

These types are called

- kinds.

Kinds of Types

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

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

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

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

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

where * (read as "star" or as "type") indicates that the type is a concrete type.

Type Constructors

Type constructors

- take types as parameters to eventually produce concrete types.

Example:

The type constructors 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 to be Int and String, respectively, the concrete types

```
Maybe Int
Either Int String
Tree Int
```

Kinds of Type Constructors

Like concrete types

- type constructors have types, called kinds, as well.

```
ghci> :k Maybe
Maybe :: * -> *

ghci> :k Either
Either :: * -> * -> *

ghci> :k Tree
Tree :: * -> *

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

Kinds of Partially Applied Type Constructors

Like functions

- also type constructors can be partially applied.

```
ghci> :k Either Int
Either Int :: * -> *

ghci> :k Either Int String
Either Int String :: *
```

Type Constructors as Functors

Reconsidering the definition of type class `Functor`

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

it becomes obvious that only

- type constructors of kind `* -> *`

can be members of type class `Functor`.

Chapter 10: Further Reading (1)

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

Motivation

Monads – A mundane approach for composing functions, for

- functional composition!

The monad approach succeeds in

- linking and composing functions

whose types are incompatible and thus inappropriate to allow their

- simple functional composition.

Monads: A Suisse Knife for Programming

Monadic programming works well for problems involving:

- Global state
  - Updating data during computation is often simpler than making all data dependencies explicit (State Monad).
- Huge data structures
  - No need for replicating a data structure that is not needed otherwise.
- Side-effects and explicit evaluation orders
  - Canonical scenario: Input/output operations (IO Monad).
- Exception and error handling
  - Maybe Monad

Illustration

Consider:

```
a-b  -- Evaluation order of a and b is not
     -- fixed. This is crucial, if input/output
     -- is involved.
```

Monads

- allow us to explicitly specify the order, in which operations are applied; this way, they bring an imperative flavour into functional programming.

```
do a <- getInt   -- Evaluation order is
   b <- getInt   -- explicitly fixed:
   return (a-b) -- first a, then b.
```

Chapter 11.1

Motivation

Setting the Stage

Consider:

```
f :: a -> b
g :: b -> c
```

Functional composition for `f` and `g` works perfectly:

```
(g . f) = g (f v)
```

where

```
(g . f) :: a -> c
```

Case Study “Debugging” (1)

Objective:

- Empowering f and g such that debug-information in terms of a string is collected and output during computation.

To this end, replace f and g by two new functions f’ and g’:

```
type DebugInfo = String

f' :: a -> (b,DebugInfo)
g' :: b -> (c,DebugInfo)
```

Unfortunately:

- f’ and g’ cannot be composed easily: Simple functional composition does not work any longer because of incompatible argument and result types of f’ and g’.

Case Study "Debugging" (2)

The below *ad hoc* composition works:

```
h v =
  let (fResult,fInfo) = f' v
      (gResult,gIinfo)
         = g' fResult in (gResult,gInfo++fInfo)
```

...but were impractical in practice as it continuously required implementing new specific composition operations.

Case Study "Debugging" (3)

Towards a more systematic approach:

- Define a new "link" function.

```
link :: (a,DebugInfo) -> (a -> (b,DebugInfo))
                                  -> (b -> DebugInfo)
link (v,s) g = let
  (gResult,gInfo) = g v in (gResult,s++gInfo)
```

The function `link` allows us to compose `f'` and `g'` comfortably again:

```
h' v = f' v 'link' g'
```

Making it Practical: `link`, `unit`, `lift`

Introduce a new identity function that is a unit for `link`, and a new lift function that makes each function working with `link`:

```
unit v = (v,"")
lift f = unit . f
```

The functions `link`, `unit`, and `lift` can now be applied in concert.

Example:

```
f v = (v, "f called. ")
g v = (v, "g called. ")

h v = f v 'link' g 'link' (\x -> (x, "done."))
```

We obtain:

```
h 5 ->> (5, "f called. g called. done.")
```

Note that functions are applied "left to right" as desired.

Case Study "Random Numbers" (1)

The library `Data.Random` provides a function

```
random :: StdGen -> (a,StdGen)
```

for computing (pseudo) random numbers.

Ordinary functions can use random numbers, if they can (additionally) manage a value of type `StdGen` that can be used by the next operation to generate a random number:

```
f :: a -> StdGen -> (b,StdGen)
```

Problem:

- How to compose functions `f` and `g`?

  ```
  f :: a -> StdGen -> (b,StdGen)
  g :: b -> StdGen -> (c,StdGen)
  ```

Case Study “Random Numbers” (2)

An *ad hoc* composition:

```
h :: a -> StdGen -> (c,StdGen)
h v gen = let
  (fResult,fGen) = f v gen in g fResult fGen
```

More appropriate:

- The trio of functions link, unit, lift.

```
link :: (StdGen -> (a,StdGen)) ->
             (a -> StdGen -> (b,StdGen)) ->
                StdGen -> (b,StdGen)
link :: g f gen = let (v,gen’) = g gen in f v gen’

unit v gen = (v,gen)
lift f     = unit . f
```

Quintessence

The previous examples enjoy

- a common structure.

This common structure can be encapsulated in a

- new (type) constructor class.

This type class will be the (constructor) class

- `Monad`.

Prospect: The Constructor Class Monad

```
data Debug a  = D (a,String)
data Random a = R (StdGen -> (a,StdGen)

class Monad m where
-- link
(>>=) :: m a -> (a -> m b) -> m b

-- link but ignore the result component of the
-- first function
(>>) :: m a -> m b -> m b

-- neutral element wrt (>>=)
return :: a -> m a
fail :: String -> m a

-- default implementation
m >> k = f >>= _ -> k
fail   = error
```

Prospect: Instance Declaration for Random

The instance declaration for type constructor Random:

```
instance Monad Random where
 (R m) >>= f = R $ \gen -> (let
                              (a,gen') = m gen
                              (R b) = f a in b gen')
 return x    = R $ \gen -> (x,gen)
```

Chapter 11.2

Constructor Class Monad

The Constructor Class Monad

Monads are instances of the constructor class Monad:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  m >> k = m >>= \_ -> k -- default implementation
  fail s = error s        -- default implementation:
                          -- represents a failing
                          -- computation that outputs
                          -- the error message s
```

...where the implementations of the monad operations (>>=), (>>), return, fail must satisfy the so-called monad laws.

The Laws of Monad

Members of the constructor class Monad must satisfy the following three laws:

```
return a >>= f                  = f a                 (ML1)
c >>= return                    = c                   (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g           (ML3)
```

Intuitively:

- return passes the value without any other effect; return is unit of (>>=).
- sequencings given by (>>=) do not depend on how they are bracketed; (>>=) is associative.

Note:

- It needs to be proven that these laws are satisfied by a concrete instance of class Monad such as trees, lists, etc. This is a proof obligation for the programmer!

The Laws of Monad in Terms of (>@>) (1)

The derived operation (>@>) makes the intuitive meaning of the monad laws more obvious; i.e. as obvious as associativity is for the (>>) operation:

```
c1 >> (c2 >> c3) = (c1 >> c2) >> c3
```

(Note: Associativity of (>>) is implied by that of (>>=).)

The operation (>@>) is defined by:

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c)
                                         -> (a -> m c)
f >@> g = \x -> (f x) >>= g
```

The Laws of Monad in Terms of (>@>) (2)

The monad laws in terms of (>@>):

```
return >@> f = f                                (ML1')
f >@> return = f                                (ML2')
(f >@> g) >@> h = f >@> (g >@> h)               (ML3')
```

Intuitively

- (ML1'), (ML2'): return is unit of (>@>).
- (ML3'): (>@>) is associative.

Note: As mentioned before, the above properties need to ensured by the instance declaration. They do not hold *per se*.

Syntactic Sugar: The do-Notation

Monadic operations

- allow to specify the sequencing of operations explicitly.

This introduces

- an imperative flavour into functional programming.

The syntactic sugar of the so-called

- do-notation

makes this flavour more explicit.

do-Notation: A Useful Notational Variant (1)

The do-notation makes composing monadic operations syntactically more concise.

Four transformation rules

- allow to convert compositions of monadic operations into equivalent (<=>) do-blocks and vice versa.

```
(R1) do e <=> e

(R2) do e1;e2;...;en <=> e1 >>= \_ -> do e2;...;en
                     <=> e1 >> do e2;...;en

(R3) do let decllist;e2;...;en <=> let decllist
                                   in do e2;...;en

(R4) do pattern <- e1;e2;...;en <=>
        let ok pattern = do e2;...;en
            ok _   = fail "..."
            in e1 >>= ok
```

do-Notation: A Useful Notational Variant (2)

A special case of the "pattern rule" (R4):

```
(R4') do x <- e1;e2;...;en <=>
          e1 >>= \x -> do e2;...;en
```

Remarks:

- (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) that 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.

- The monad laws using the monadic operations:

```
return a >>= f                  = f a                 (ML1)
c >>= return                    = c                   (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g (ML3)
```

- The monad laws using the 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)
```

Quintessence: The Constructor Class Monad

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b
 return :: a -> m a
 (>>)   :: m a -> m b -> m b
 fail   :: String -> m a

 m >> k = m >>= \_ -> k  -- default implementation
 fail s = error s        -- default implementation
```

Intuitively:

Monad operations

- describe actions with side effects.
- allow to fix the order of evaluation steps.
- support an imperative-like programming style w/out breaking the functional paradigm.

Quintessence: Monadic Operations

Intuitively

- `(>>=)`: The sequence operator (read as then (according to Simon Thompson) or bind (according to Paul Hudak)), or – maybe – as link.
- `return`: Returns a value w/out any other effect.
- `(>>)`: From `(>>=)` derived sequence operator (read as sequence (according to Paul Hudak)).
- `fail`: Exception and error handling.

Useful Supporting Functions for Monads

```
sequence    :: Monad m => [m a] -> m [a]
sequence    =  foldr mcons (return [])
                 where mcons p q = do l  <- p
                                      ls <- q
                                      return (l:ls)

sequence_   :: Monad m => [m a] -> m ()
sequence_   =  foldr (>>) (return ())

mapM        :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as   =  sequence (map f as)

mapM_       :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as  =  sequence_ (map f as)

(=<<)       :: Monad m => (a -> m b) -> m a -> m b
f =<< x     =  x >>= f
```

A Law linking Classes Monad and Functor

Type constructors that are an instance of both

- class Monad and class Functor

must satisfy the law:

```
fmap g xs  =  xs >>= return . g                        (MFL)
         ( =  do x <- xs; return (g x) )
```

Chapter 11.3

Predefined Monads

Predefined Monads

A selection of predefined monads in Haskell:

- Identity monad
- List monad
- Maybe monad
- State monad

The Identity Monad (1)

The identity monad, conceptually the simplest monad:

```
newtype Id a = Id a

instance Monad Id where
  (Id x) >>= f  =  f x
  return        =  Id
```

Note:

- `(>>)` and `fail` are implicitly defined by their default implementations.

The Identity Monad (2)

Remarks:

- The identity monad maps a type to itself.
- It represents the trivial state, in which no actions are performed, and values are returned immediately.
- It is useful because it allows to specify computation sequences on values of its type.
- The operation `(>@>)` becomes for the identity monad forward composition of functions, i.e., `(>.>)`:

```
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
g >.> f = f . g
```

- Forward composition of functions `(>.>)` is associative with unit `id`.

The List Monad (1)

The list monad:

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
  fail s   = []
```

where concat is from the Standard Prelude:

```
concat     :: [[a]] -> [a]
concat lss = foldr (++) [] lss
```

The List Monad (2)

The list monad can equivalently be defined by:

```
instance Monad [] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  [] >>= f = []
  return x = [x]
  fail s   = []
```

Note:

- For the list monad the monadic operations (>>=) and `return` have the types:

```
(>>=)  :: [a] -> (b -> [b]) -> [b]
return :: a -> [a]
```

The List Monad (3)

The list monad is closely related to list comprehension:

```
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)]
```

Hence, the following notations are equivalent:

```
[(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!

The List Monad (4)

List comprehension: Syntactic sugar for monadic syntax.

We have:

```
[f x | x <- xs]      <=> do x <- xs; return (f x)

[a | a <- as, p a] <=>
  do a <- as; if (p a) then return a else fail ""
```

The Maybe Monad (1)

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
```

Remark:

- The Maybe monad is useful for computation (sequences) that might produce a result, but might also produce an error.

The Maybe Monad (2)

For the Maybe monad the monadic operations (>>=) and return have the types:

```
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a
```

The Maybe type is also a predefined member of the Functor class:

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

The Maybe Monad (3)

Composing functions like

```
f :: Int -> Int
g :: Int -> Int
x :: Int
```

in `g (f x)` while assuming that the evaluation of `f` and `g` may fail, is possible by embedding the computation into the `Maybe` type:

```
case (f x) of
  Nothing -> Nothing
  Just y  -> case (g y) of
               Nothing -> Nothing
               Just z  -> z
```

Though possible, this is "inconvenient."

The Maybe Monad (4)

Embedding gets a lot easier by exploiting the membership of the Maybe type in the Maybe monad:

```
f x >>= \y ->  g y >>= \z -> return z
```

which is equivalent to:

```
do y <- f x
   z <- g y
   return z
```

...the "nasty" error check is "hidden" in the Maybe monad.

The Maybe Monad (5)

Note that

```
f x >>= \y ->  g y >>= \z -> return z
```

can also 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
```

This way, `g (f x)` gets `f x >>= g`.

The Maybe Monad (6)

Another possibility to better cope with `(g . f) x` were to introduce the function:

```
composeM :: Monad m => (b -> m c) ->
                              (a -> m b) -> (a -> m c)
(g 'composeM' f) x = f x >>= g
```

Using `composeM` we obtain:

`(g . f) x` gets `(g 'composeM' f) x`

Note:

- Both this and the previous handling of embedding the function composition of `g` and `f` into the `Maybe` type preserve the original notation of composing `g` and `f` in an almost 1-to-1 kind.

The State Monad (1)

Objective:

- Modelling of programs with global (internal) state and side effects by means of
  - functions that applied to an initial state yield a final state as part of the overall result of the compuation.

The (resp. a) state monad:

```
newtype State s a = St (s -> (s,a))

instance Monad (State s) where
  return x     = St (\s -> (s,x)) -- The identity
  (St m) >>= f =                  -- on states!
     St (\s -> let (s1,x) = m s   -- m applied to
                   St f'  = f x   -- s yields s1
               in f' s1)          -- and x to which
                                  -- then f is
                                  -- applied to.
```

The State Monad (2)

Intuitively

State transformers

- model and transform global (internal) states.
- are (in this setting) mappings of the type `s -> (s,a)`.
- map an initial state to a pair consisting of a (possibly modified) final state and another result component of type `a`.

The State Monad (3)

A slightly different definition of the state monad for S some suitable type:

```
data SM a = SM (S -> (S,a))

instance Monad SM where
  return a
     = SM (\s -> (s,a))
  SM sm0 >>= fsm1
     = SM $ \s0 ->
         let (s1,a1) = sm0 s0
             SM sm1  = fsm1 a1
             (s2,a2) = sm1 s1
         in (s2,a2)
```

Predefined Monads

There are many more predefined monads in Haskell:

- Writer monad
- Reader monad
- Failure monad
- ...
- Input/output monad

The Input/Output Monad (1)

The IO monad:

```
instance Monad IO where
(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

Intuitively:

- `(>>=)`: If `p` and `q` are commands, then `p >>= q` is the command that first executes `p`, yielding thereby the return value `x` of type `a`, and then executes `q x`, thereby yielding the return value `y` of type `b`.
- `return`: Generates a return value w/out any input/output action.

The Input/Output Monad (2)

It is worth noting:

- The IO monad is similar in spirit to the state monad: It passes around the "state of the world."

In more detail:

For a given suitable type World

- whose values represent the current state of the world

the notion of an interactive program, i.e., an IO-program, can be represented by a function of type

- World -> World

which may be abbreviated as:

```
type IO = World -> World
```

The Input/Output Monad (3)

In general:

- Interactive programs do not only modify the state of the world but may also return a result value, e.g., echoing a character that has been read from a keyboard.

This suggests to change the type of interactive programs to

```
type IO = World -> (a, World)
```

Chapter 11.4

Constructor Class MonadPlus

The Monad MonadPlus

...for members of Monad with Null and Plus operation:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

The Laws of MonadPlus

Members of the constructor class MonadPlus must satisfy in addition to the monad laws laws for the Null and Plus operations:

Two laws for the Null operation:

```
m >>= (\x -> mzero) = mzero               (MPL1)
mzero >>= m         = mzero               (MPL2)
```

Two laws for the Plus operation:

```
m 'mplus' mzero = m                       (MPL3)
mzero 'mplus' m = m                       (MPL4)
```

Note:

- As for Functor and Monad, proving the validity of the above laws for an instance of class MonadPlus is a proof obligation for the programmer.

Instances of MonadPlus

Instance declarations for the Maybe and [] types for the class MonadPlus:

```
instance MonadPlus Maybe where
  mzero                 = Nothing
  Nothing 'mplus' ys = ys
  xs 'mplus' ys         = xs

instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Note:

- List concatenation (++) is a special case of the mplus operation.
- IO is not an instance of MonadPlus because of the missing null element.

Chapter 11.5

Monadic Programming

Outline

We will consider three case studies for illustration:

- Case study I: Summing labels of a tree.
- Case study II: Replacing the leaf labels of a tree by leaf labels of another type.
- Case study III: Replacing the labels of a tree by the number of occurrences of this label in the tree.

Case Study I

Given:

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

Objective:

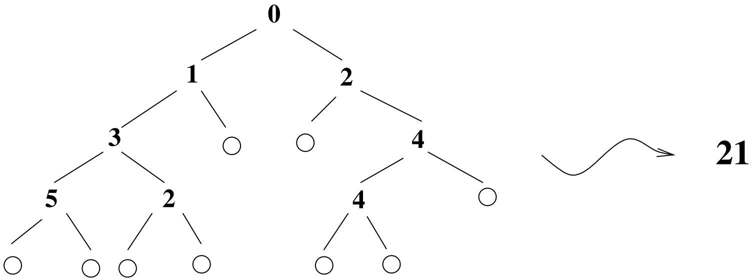
- Write a function that computes the sum of the values of all labels of a tree of type `Tree Int`.

Means:

Opposing two different functional approaches:

- A classical functional approach w/out monads
- A functional approach w/ monads.

Illustration



A Functional Approach w/out Monads

1st Approach: No monads

```
sTree :: Tree Int -> Int
sTree Nil              = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```

Note:

- The order of the evaluation is not fixed (degrees of freedom!)

A Functional Approach w/ Monads

2nd Approach: Using the identity monad `Id`

```
sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2) = do num <- return n
                            s1  <- sumTree t1
                            s2  <- sumTree t2
                            return (num + s1 + s2)
```

Note:

- The order of the evaluation is explicitly fixed (no degrees of freedom!)

The Identity Monad

Recall the identity monad:

```
data Id a = Id a

instance Monad Id where
  (>>=) (Id x) f = f x
  return         = Id
```

Opposing the Two Approaches

Comparing the two approaches w/ and w/out monads, we observe:

- Unlike sTree, function sumTree has an "imperative" flavour very similar to the sequential sequence of (imperative) assignments:

Imperative	Monadic
`num := n;`	`do num <- return n`
`s1  := sumTree t1;`	`   s1  <- sumTree t1`
`s2  := sumTree t2;`	`   s2  <- sumTree t2`
`return (num + s1 + s2);`	`   return (num + s1 + s2)`

Another Functional Approach w/ Monads

3rd Approach: Using monad Id and an extraction function

```
extract :: Id a -> a
extract (Id x) = x
```

Using `extract` we get a function of type `Tree Int -> Int`:

```
extract . sumTree :: Tree Int -> Int
```

Example:

```
(extract . sumTree)
  (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil))
 ->>
extract (sumTree
  (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil)))
 ->>
extract (Id 15) ->> 15
```

Case Study II

Given:

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

Objective:

- Replace the labels of the leafs that are supposed to be of type `Char` by continuous natural numbers.

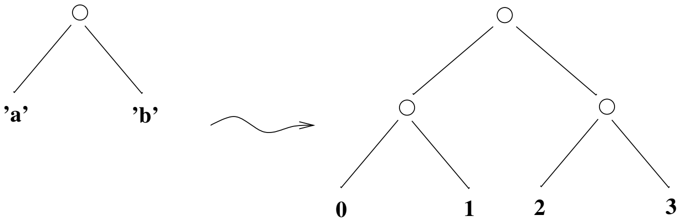
Illustration

Let `test` be defined by

```
test = let t = Branch (Leaf 'a') (Leaf 'b')
       in label (Branch t t)
```

Then `test` shall be transformed to:

```
Branch (Branch (Leaf 0) (Leaf 1))
       (Branch (Leaf 2) (Leaf 3))
```



A Functional Approach w/out Monads

1st Approach: No monads

```
label   :: Tree a -> Tree Int
label t = snd (lab t 0)

lab :: Tree a -> Int -> (Int, Tree Int)
lab (Leaf a) n
    = (n+1, Leaf n)
lab (Branch t1 t2) n
    = let (n1,t1') = lab t1 n
          (n2,t2') = lab t2 n1
      in (n2, Branch t1' t2')
```

Note:

- Simple but passing the value n through the incarnations of lab is "intricate."

A Functional Approach w/ Monads (1)

2nd Approach: Using the state monad

```
newtype Label a = Label (Int -> (Int,a))
```

..."matches" the pattern of the state monad SM.

We define:

```
instance Monad Label where
  return a
    = Label (\s -> (s,a))
  Label lt0 >>= flt1
    = Label $ \s0 ->
        let (s1,a1)   = lt0 s0
            Label lt1 = flt1 a1
        in lt1 s1
```

Note: The $-operator in the definition of `(>>=)` can be dropped, if the expression `\s0 -> let ...  in lt1 s1` is bracketed.

A Functional Approach w/ Monads (2)

This allows solving the renaming of labels as follows:

```
mlabel   :: Tree a -> Tree Int
mlabel t = let Label lt = mlab t
           in snd (lt 0)

mlab :: Tree a -> Label (Tree Int)
mlab (Leaf a)
     = do n <- getLabel
          return (Leaf n)
mlab (Branch t1 t2)
     = do t1' <- mlab t1
          t2' <- mlab t2
          return (Branch t1' t2')

getLabel :: Label Int
getLabel = Label (\n -> (n+1,n))
```

A Functional Approach w/ Monads (3)

Let mtest be defined by

```
mtest = let t = Branch (Leaf 'a') (Leaf 'b')
        in mlabel (Branch t t)
```

Then we get:

- mlabel applied to

  `Branch (Leaf 'a') (Leaf 'b')`

  yields as desired:

  ```
  Branch (Branch (Leaf 0) (Leaf 1))
         (Branch (Leaf 2) (Leaf 3))
  ```

Case Study III

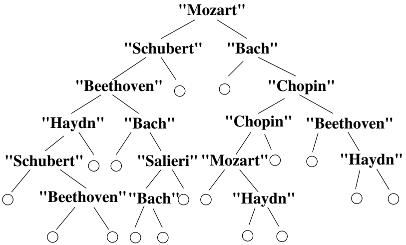
Given:

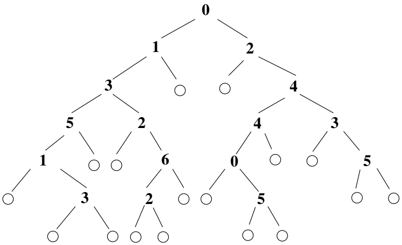
```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Objective:

- Replace labels of equal value that are supposed to be of type `String` by the same natural number.

Illustration





A Functional Approach w/ Monads (1)

Ultimate Goal: A function `numTree` of type

```
numTree :: Eq a => Tree a -> Tree Int
```

solving this task with monadic programming using the state monad.

In order to eventually arrive at this function we start with:

```
numberTree :: Eq a => Tree a -> State a (Tree Int)
numberTree Nil = return Nil
numberTree (Node x t1 t2) =
  do num <- numberNode x
     nt1 <- numberTree t1
     nt2 <- numberTree t2
     return (Node num nt1 nt2)
```

A Functional Approach w/ Monads (2)

Next, we are storing pairs of the form

<string>,<number of occurrences>)

in a table of type:

```
type Table a = [a]
```

In particular:

The table

```
[True, False]
```

encodes that the value `True` is associated with `0` and `False` with 1.

A Functional Approach w/ Monads (3)

Defining the state monad we consider:

```
data State a b = State (Table a -> (Table a, b))

instance Monad (State a) where
 (State st) >>= f
    = State (\tab -> let
                     (newTab,y)    = st tab
                     (State trans) = f y
                     in
                     trans newTab)
  return x = State (\tab -> (tab,x))
```

Intuitively:

- Values of type b: Result of the monadic operation.
- Update of the table: Side effect of the monadic operation.

A Functional Approach w/ Monads (4)

Defining the missing function numberNode:

```
numberNode :: Eq a => a -> State a Int
numberNode x = State (nNode x)

nNode :: Eq a => a -> (Table a -> (Table a, Int))
nNode x table
  | elem x table = (table,       lookup x table)
  | otherwise    = (table++[x], length table)
-- nNode yields the position of x in the table:
-- via lookup, if stored in the table; after
-- adding x to the table via length otherwise

lookup :: Eq a => a -> Table a -> Int
lookup ... (still to complete)
```

A Functional Approach w/ Monads (5)

Putting the pieces together, we get for `exampleTree :: Eq a => Tree a`:

```
numberTree exampleTree :: State a (Tree Int)
```

Using an extraction function we get now the desired implementation of the function `numTree` of type `numTree :: Eq a => Tree a -> Tree Int`:

```
extract :: State a b -> b
extract (State st) = snd (st [])

numTree :: Eq a => Tree a -> Tree Int
numTree = extract . numberTree
```

Chapter 11.6

Monadic Input/Output

Handling Input/Output so Far

The programs we considered so far, handle input/ouput monolithicly, in a way that resembles

- batch processing.

Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S.245

In fact, there is no interaction between a program and a user:

- All input data must be provided at the very beginning.
- Once called there is no opportunity for the user to react on a program's response and behaviour.

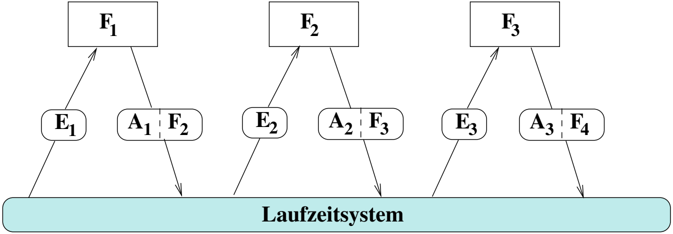
Handling Input/Output Henthforth

Our Objective:

Modifying the handling of input/ouput such that programs become and behave like

- (sequentially) composed dialogue components

while preserving referential transparency as far as possible.



Peter Pepper. *Funktionale Programmierung.*
Springer–Verlag, 2003, S.253

It is worth noting

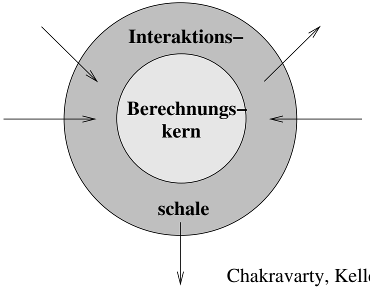
As illustrated by the previous figure, input/output is

- a major source for side effects in a program: e.g., each read statement like `read` will usually yield a different value for each call, i.e. referential transparency is lost.

Monadic Input/Output in Haskell

Conceptually, a Haskell program consists of

- a computational core and
- an interaction component.



Chakravarty, Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004, S.89

Monadic Input/Ouput

The monad concept of Haskell allows to

- distinguish (and conceptually separate) functions that belong to the
  - computational core (pure functions)
  - interaction component (impure functions, i.e. having side effects).

  by assigning different types to them:

  ⇝ `Int`, `Real`, `String`,... vs. `IO Int`, `IO Real`, `IO String`,... where the type constructor `IO` is an instance of `Monad`.

- specify the evaluation order of functions of the interaction component (i.e., of basic input/output primitives provided by Haskell) by explicitly using the features of monadic programming.

Recall Chapter 11.3

The Input/Output Monad (1)

The IO monad:

```
instance Monad IO where
(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

Intuitively:

- (>>=): If p and q are commands, then p >>= q is the command that first executes p, yielding thereby the return value x of type a, and then executes q x, thereby yielding the return value y of type b.
- return: Generates a return value w/out any input/output action.

Recall Chapter 11.3

The Input/Output Monad (2)

It is worth noting:

- The `IO` monad is similar in spirit to the state monad: It passes around the "state of the world."

In more detail:

For a given suitable type `World`

- whose values represent the current state of the world

the notion of an interactive program, i.e., an IO-program, can be represented by a function of type

- `World -> World`

which may be abbreviated as:

```
type IO = World -> World
```

Recall Chapter 11.3

The Input/Output Monad (3)

In general:

- Interactive programs do not only modify the state of the world but may also return a result value, e.g., echoing a character that has been read from a keyboard.

This suggests to change the type of interactive programs to

```
type IO = World -> (a, World)
```

Typical Interaction Examples (1)

A simple question/response interaction with the 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 ++ "!")
```

Typical Interaction Examples (2)

Input/output from/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
```

Typical Interaction Examples (3)

Note the relationship of the do-notation

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
```

and the monadic operations:

```
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```

Note also the (subtle) difference in the result types:

```
Main>putStr ('a':('b':('c':[])))
            ->> abc :: IO ()
```

```
Main>putChar (head ['x','y',
            ->> x :: IO ()
```

but

```
Main>('a':('b':('c':[])))
            ->> "abc" :: [Char]
```

```
Main>head ['x','y','z']
            ->> 'x' :: Char
```

```
Main>print "abc"
            ->> "abc" :: IO ()
```

```
Main>print 'x'
            ->> 'a' :: IO ()
```

More Examples (1)

The output command sequence

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
```

...is equivalent to:

```
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```

More Examples (2)

It is worth noting:

From

```
(>>) :: Monad m => m a -> m b -> m b
```

and

```
writeFile "testFile.txt"
          "Hello File System!" :: IO ()
putStr "Hello World!"          :: IO ()
```

...we conclude for our example that `m = IO`, `a = ()`, and `b = ()`. Overall, we thus obtain:

```
(>>) :: IO () -> IO () -> IO ()
```

More Examples (3)

Illustrating local declarations within do-constructs:

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        let rev1 = reverse line1
        let rev2 = reverse line2
        putStrLn rev2
        putStrLn rev1
```

is equivalent to:

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        putStrLn (reverse line2)
        putStrLn (reverse line1)
```

Summing up (1)

Overall, the monadic handling of input/output in Haskell renders possible:

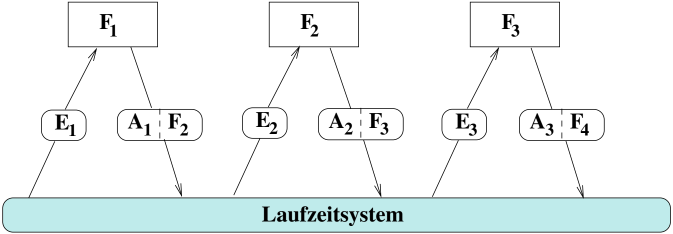
The shift from

- "batch-like" input/output processing that works exclusively by pure functions of the computational core as illustrated below

Peter Pepper. *Funktionale Programmierung*. Springer–Verlag, 2003, S.245

Summing up (2)

...to an interactive, dialogue-oriented input/output processing w/out breaking the functional paradigm (keyword: referential transparency!)

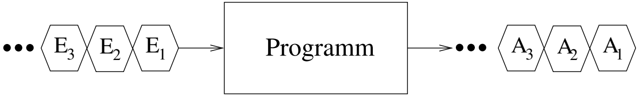


Peter Pepper. *Funktionale Programmierung*. Springer–Verlag, 2003, S.253

Stream-based Input/Output (1)

Early versions of Haskell foresaw a stream-based handling of input/output:

- Stream-based considering programs functions on streams:
  `IOprog ::  String -> String`



Peter Pepper. *Funktionale Programmierung*. Springer–Verlag, 2003, S.271

Input/output streams on terminals, file systems, printers,...

Stream-based Input/Output (2)

Advantages and disadvantages:

- Stream-based input/output handling for languages with
  - eager semantics:
    - there is no real stream model (the input must completely be provided and consumed at the beginning and must thus be finite); hence, input/output is limited to a batch- or stack-like processing.
  - lazy semantics:
    - Interactions are possible; thanks to lazy evaluation inputs/outputs are always in "proper" order.
    - But: the causal and temporal relationship between input and output is often "obscure"; special synchronization might be used to overcome that.
    - Overall: streambased input/output reaches its limit when switching to graphical user interfaces and random access to files.

ML-Style Input/Output

The ML-style of handling input/output is

- a Unix-like handling of display, keyboard, etc. as files: `std_in, std_out, open_in, open_out, close_in,...`

Advantages and disadvantages:

- The handling is simple but at the cost of anomalies like those discussed in LVA 185.A03; in particular, referential transparency is lost.

Last but not least

Input/output handling in functional languages is an important research topic:

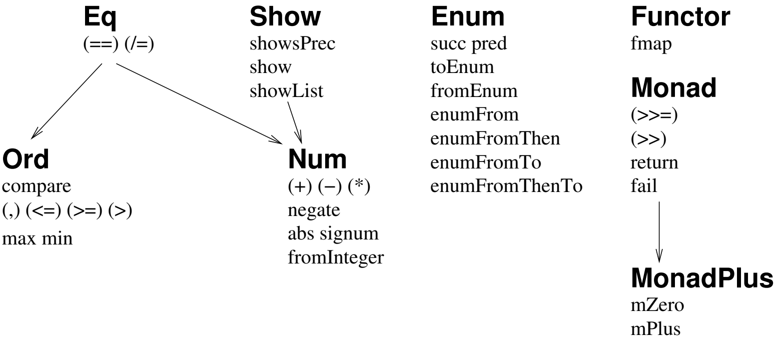
- Andrew D. Gordon. Functional Programming and Input/Output. British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1992.

Chapter 11.7

A Fresh Look at the Haskell Class Hierarchy

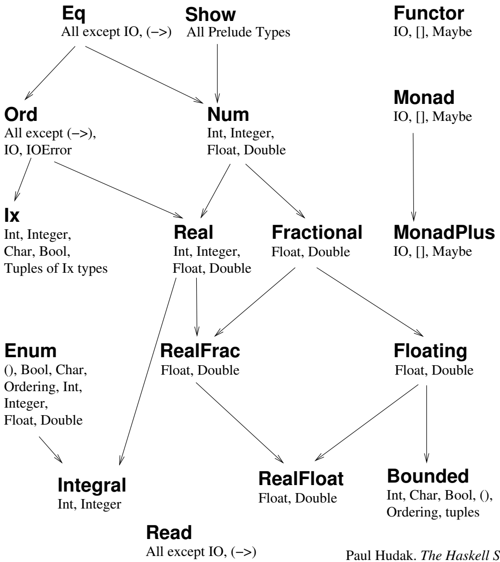
A Section of the Haskell Class Hierarchy (1)

...including the constructor classes `Monad`, `MonadPlus`, and `Functor`:



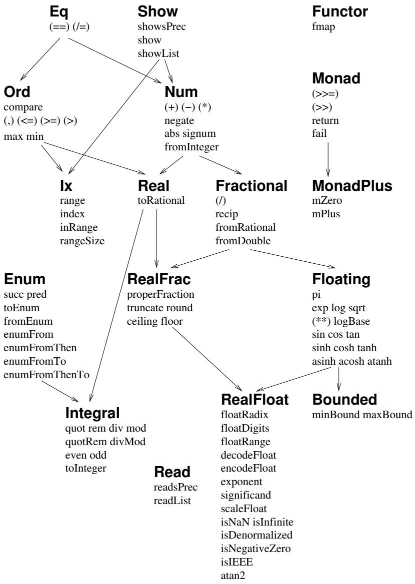
Fethi Rabhi, Guy Lapalme *Algorithms.*
Addison−Wesley, 1999, Figure 2.4, p.46

A Section of the Haskell Class Hierarchy (2)



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

A Section of the Haskell Class Hierarchy (3)



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

Selected Types and their Class Membership

Type	Instance of	Derivation
()	Read	Eq Ord Enum Bounded
[a]	Read Functor Monad	Eq Ord
(a,b)	Read	Eq Ord Bounded
(−>)		
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 Monad	
IOError	Eq	
Maybe	Functor Monad	Eq Ord Read
Ordering		Eq Ord Enum Read Bounded
Ratio	RealFrac Read	

Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison−Wesley, 1999, Table 2.4, p. 47

Last but not least (1)

Monads – where does the term come from?

Monads, a term that

- has already been used by Gottfried Wilhelm Leibniz as a counterpart to the term "atom."
- has been introduced into programming language theory by Eugenio Moggi in the realm of category theory as a means for describing the semantics of programming languages:

  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.

Last but not least (2)

Monads, a term that

- has become popular in the world of functional programming (but w/out the background from category theory), especially because monads (Philip Wadler, 1992)
  - allow to introduce some useful aspects of imperative programming into functional programming,
  - are well suited for integrating input/output into functional programming, as well as for many other application domains,
  - provide a suitable interface between functional programming and programming paradigms with side effects, in particular, imperative and object-oriented programming.

  without breaking the functional paradigm!

Chapter 11: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17, Monaden)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
- 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.
- 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)

Chapter 11: Further Reading (2)

- 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)
- 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.

Chapter 11: Further Reading (3)

- 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)
- 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)

Chapter 11: Further Reading (4)

- 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)
- 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.

Chapter 11: Further Reading (5)

- 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).
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.2, Monads)
- Michael Spivey. *A Functional Theory of Exceptions*. Science of Computer Programming 14(1):25-42, 1990.

Chapter 11: Further Reading (6)

- 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.
- 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)

Chapter 11: Further Reading (7)

- 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.
- 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.

Chapter 11: Further Reading (8)

- 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.

Chapter 12

Arrows

Motivation

The higher-order type (constructor) class

- generalizes the type class `Monad`.

and provides an even more general concept for

- composing functions.

that is particularly useful for

- functional reactive programming (cp. Chapter 15).

The Constructor Class Arrow

Arrows are instances of the constructor class Arrows:

```
class Arrow a where
  pure  :: (b -> c) a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

Note:

- pure allows embedding of ordinary functions into the constructor class Arrow.
- (>>>) serves the composition of computations.
- first has as an analogue on the level of ordinary functions the function firstfun with `firstfun f = \(x,y) -> (f x, y)`

The Laws of Arrow

Members of the constructor class `Arrow` must satisfy the following nine 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
```

Creating Instances of Class Arrow

Ordinary functions as instance of constructor class Arrow:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f × id
```

Note:

- The function first could also be defined by:
  `first f = \(b,d) -> (f b, d)`

Useful Supporting Functions (1)


```
(×) :: (a -> a') -> (b -> b') -> (a,b) -> (a',b')
(f × g) (a,b) = (f a, g b)

assoc :: ((a,b),c) -> (a,(b,c))
assoc~(~(x,y),z) = (x,(y,z))

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = pure swap >>> first f >>> pure swap

swap :: (a,b) -> (b,a)
swap~(x,y) = (y,x)
```

Useful Supporting Functions (2)

...related to the 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,b)) >>> f *** g

idA :: Arrow a => a b b
idA = pure id
```

Background and Motivation (1)

Notions of computation:

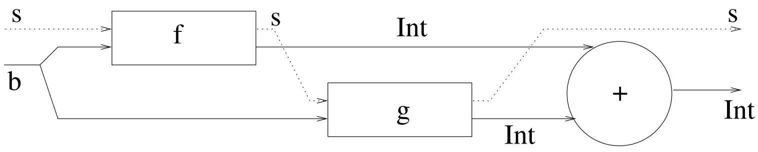
```
add :: (b -> Int) -> (b -> Int) -> (b -> Int)
add f g z = f z + g z
```

Background and Motivation (2)

Generalizing add to state transformers:

```
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:



Background and Motivation (3)

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 ]
```

Background and Motivation (4)

Generalizing add to mapping 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
```

Background and Motivation (5)

Generalizing add to simple automata:

```
newtype Auto i o = A (i -> (o, Auto i o))

addAuto :: Auto b Int -> Auto b Int -> Auto b Int
addAuto (A f) (A g)
   = A (\z -> let (x,f') = f z
                  (y,g') = g z
              in (x+y), addAuto f' g'))
```

All together, this

- allows modelling of synchronous circuits.

Background and Motivation (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 programs.

Back to the Examples (1)

- The preceding examples have in common that there is a type $A \rightsquigarrow B$ of computations, where inputs of type A are transformed into outputs of type B.
- `Arrows` yield a suffiently general interface to describe these commonalities uniformly and to encapsulate them in a class.

Back to the Examples (2)

Implementing the preceding examples as instances of the class Arrow:

```
newtype State s i o = ST ((s,i) -> (s,0))

newtype NotDet i o  = ND (i -> [o])

newtype MapTrans s i o = MT ((s -> i) -> (s -> o))

newtype Auto i o = A (i -> (o, Auto i o))
```

Back to the Examples (3)

State transformers:

```
instance Arrow (State s) where
  pure f         = ST (id x f)
  ST f >>> ST g = ST (g . f)
  first (ST f)   = ST (assoc . (f x id) . unassoc)

unassoc              :: (a,(b,c)) -> ((a,b),c)
unassoc~(x,~(y,z)) = ((x,y),z)
```

Back to the Examples (4)

Non-determinism:

```
instance Arrow NonDet where
 pure f          = ND (\b -> [f b])
 ND f >>> ND g = ND (\b -> [d | c <- f b, d <- g c])
 first (ND f)  = ND (\(b,d) -> [(c,d) | c <- f b])
```

Back to the Examples (5)

Mapping transformers:

```
instance Arrow (MapTrans s) where
 pure f         = MT (f .)
 MT f >>> MT g = MT (g . f)
 first (MT f)   = MT (zipMap . (f x id) . unzipMap)

zipMap     :: (s -> a, s -> b) -> (s -> (a,b))
zipMap h s = (fst h s, snd h s)

unzipMap   :: (s -> (a,b)) -> (s -> a, s -> b)
unzipMap h = (fst . h, snd . h)
```

Back to the Examples (6)

Simple automata:

```
instance Arrow Auto where
 pure f      = A (\b -> (f b, pure f)
 A f >>> A g = A (\b -> let (c,f') = f b
                            (d,g') = g c
                        in (d, f' >>> g')))
 first (A f) = A (\(b,d) -> let (c,f') = f b
                            in ((c,d),first f'))
```

Back to the Examples (7)

Generalization

Consider the general combinator:

```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> pure (uncurry (+))
```

It is worth noting:

- Each of the considered variants of `add` results as a specialization of `addA` with the corresponding arrow-type.

Summing up

- Arrow-combinators operate on "computations", not on values. They are point-free in distinction to the "common case" of functional programming.
- 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.

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.

Part V

Applications

Chapter 13

Parsing

Parsing

Parsing: Lexical and syntactical analysis

- Combinator parsing
- Monadic parsing

Lexical and Syntactical Analysis

...in the following summarized as parsing.

Parsing

- an(other) application of functional programming often used to demonstrate its power and elegance.
- enjoys a long history. As an example of early work see e.g.:
  - William H. Burge. Recursive Programming Techniques. Addison-Wesley, 1975.

Functional Implementation Approaches for Parsing

Two conceptually different implementation approaches:

- Combinator parsing (higher-order functions parsing)
  $\rightsquigarrow$ recursive descent 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.

The presentation here

...is based on:

- Chapter 17
  Simon Thompson. Haskell – The Craft of Functional Programming, Addison-Wesley/Pearson, 2nd edition, 1999.
- Graham Hutton, Erik Meijer. Monadic Parsing in Haskell. Journal of Functional Programming 8(4):437-444, 1998.

Parsing informally

The basic problem:

- Read a sequence of objects of type `a`.
- Extract from this sequence an object or a list of objects of type `b`.

Illustrating Example: Parsing of Expressions

Consider:

- Expressions

```
data Exp = Lit Int | Var Name | Op Ops Exp Exp
data Ops  = Add | Sub | Mul | Div | Mod
```

`Op Mul (Op Add (Lit 2) (Lit 3)) (Lit 3)`

`corresponds to  ((2+3)*3)`

The parsing task to be solved:

- Read an expression of the form `((2+3)*5)` and yield/"extract" the corresponding expression of type `Exp`.

  (Note: This can be considered the reverse of the `show` function. It is similar to the derived `read` function, but differs in the arguments it takes (expressions of the form `((2+3)*5)` vs. expressions of the form `Op Mul (Add (Lit 2) (Lit 3)) (Lit 5)`).

Towards the Type of a Parser Function (1)

What shall be the type of a parsing function?

Naive specification of the type of a parser function:

```
type BSParse1 a b = [a] -> b

-- Parser   Input        Expected Output
bracket     "(xyz" ->> '('
number      "234"  ->> 2 or 23 or 234 ?
bracket     "234"  ->> no result, failure?
```

Open issues to be answered:

How shall the parser behave if there

- are multiple results?
- is a failure?

Towards the Type of a Parser Function (2)

First refinement of the type of a parser function:

```
type BSParse2 a b = [a] -> [b]

-- Parser   Input        Expected Output
bracket     "(xyz" ->> ['(']
number      "234"  ->> [2, 23, 234]
bracket     "234"  ->> []
```

Open issue to be answered:

- What shall the parser do with the remaining input?

The Type of a Parser Function

The final specification of the type of a parser function:

```
type Parse a b = [a] -> [(b,[a])]

-- Parser   Input       Expected Output
bracket     "(xyz" ->> [('(', "xyz")]
number      "234"  ->> [(2,"34"), (23,"4"), (234,"")]
bracket     "234"  ->> []
```

Remarks and Conventions

It is worth noting:

- The capability of delivering multiple results enables the analysis of ambiguous grammars

  $\rightsquigarrow$ list of successes technique

- Each element in the output list represents a successful parse.

Convention:

- Delivery of the empty list: Signals failure of the analysis.
- Delivery of a non-empty list: Signals success of the analysis; each element of the list is a pair, whose first component is the identified object (token) and whose second component is the input not yet considered.

Chapter 13.1

Combinator Parsing

Basic Parsers (1)

Primitive, input-independent parsing functions:

- The always failing parsing function

```
none :: Parse a b
none inp = []
```

- The always successful parsing function

```
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

Remark:

- The none parser always fails. It does not accept anything.
- The succeed parser does not consume its input. In BNF-notation this corresponds to the symbol ε representing the empty word.

Basic Parsers (2)

Primitive, input-dependent parsing functions:

- Recognizing single objects (token):

```
token :: Eq a => a -> Parse a a
token t (x:xs)
  | t == x      =  [(t,xs)]
  | otherwise   =  []
token t []      =  []
```

- Recognizing single objects satisfying a particular property:

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x         =  [(x,xs)]
  | otherwise   =  []
spot p []       =  []
```

Simple Applications of Basic Parsers

Application:

```
bracket = token '('
dig     = spot isDigit

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

Note: token can be defined using spot

```
token t = spot (== t)
```

Intuition and Motivation for Combining Parsers

...obtaining (more) complex (re-usable) parsing functions:

- Combinator Parsing

Objective

- Building a library of higher-order polymorphic functions, which are then used to construct parsers.

Combining Parsers – Alternatives

1) Composition of parsers as alternatives:

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

Underlying intuition:

- An expression, e.g., is either a literal, or a variable or an operator expression.

  Example:

  `(bracket 'alt' dig) "234" ->> [] ++ [(2,"34")]`

  ⇝ The `alt` parser combines the results of the parses given by the parsers `p1` and `p2`.

Combining Parsers – Sequential Composition

2) Sequential composition of parsers:

```
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 ]
```

Underlying intuition:

- An operator expression starts with a bracket followed by a number.

Combining Parsers – Sequential Composition

Example:

Because of `number "24(" ->> [(2,"4("), (24,"(")]` we obtain:

```
(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 "(" ]
```

Because of `bracket "(" ->> [('(',"")]` we finally get:

```
 ->> [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
 ->> [ ((24,'('), "") ]
```

Combining Parsers – Transformation

3) Transformation by parsers:

⇝ transform the item returned by the parser, e.g., build something from it.

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```

Example: Note, `digList` returns a list of numbers and shall be embedded such that the number represented by it is returned.

```
(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") ]
```

Universal Parser Basis

The Clou:

The

- basic parsers

together with the combinators

- `alt`
- `(>*>)`
- `build`

constitute a universal "parser basis," i.e., allow to build any parser which might be desired.

Example: A Parser for a List of Objects

We suppose to be given a parser recognizing single objects:

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```

Intuition:

- A list can be empty.
  ⇝ this is recognized by the parser `succeed []`.
- A list can be non-empty, i.e., it consists of an object followed by a list of objects.
  ⇝ this is recognized by the combined parser `p >*> list p`, where we use `build` to turn a pair `(x,xs)` into the list `(x:xs)`.

Summing up

...on combining parsers (parser combinators):

- Parsing functions in the above fashion are structurally similar to grammars in BNF-form. For each operator of the BNF-grammar there is a corresponding (higher-order) parsing function.
- These higher-order functions combine simple(r) parsing functions to (more) complex parsing functions.
- They are thus also 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).

Summary of the Universal Parser Basis (1)

Priority of the sequence operator

```
infixr 5 >*>
```

Parser type

```
type Parse a b = [a] -> [(b,[a])]
```

Input-independent parsing functions

```
none :: Parse a b
none inp = []

succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

Summary of the Universal Parser Basis (2)

Recognizing single objects

```
token :: Eq a => a -> Parse a a
token t = spot (==t)
```

Recognizing single objects satisfying a particular property

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x         =  [(x,xs)]
  | otherwise   =  []
spot p []       =  []
```

Summary of the Universal Parser Basis (3)

Alternatives

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

Sequences

```
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp,
                    (z,rem2) <- p2 rem1 ]
```

Transformation

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```

1st Application of the Universal Parser Basis

Example

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```

2nd Application of the Universal Parser Basis

Back to the initial example – a parser for expressions.

We consider expressions of the form:

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Add (Lit 2) (Lit 3)     corresponds to 2+3
```

where the following convention shall hold:

- Literals: 67, ∼89, etc., where ∼ is used for unary minus.
- Names: the lower case characters from 'a' to 'z'.
- Applications of the binary operations ...$+, *, -, /, \%$, where % is used for mod and / for integer division.
- Expressions are fully bracketed; white space is not permitted.

A Parser for Expressions (1)

The parser

```
parser :: Parse Char Expr
parser = litParse 'alt' nameParse 'alt' opExpParse
```

... consists of three parts corresponding to the three sorts of expressions.

Part I: Parsing names of variables

```
nameParse :: Parse Char Expr
nameParse = spot isName 'build' Name

isName :: Char -> Bool
isName x = ('a' <= x && x <= 'z')
```

A Parser for Expressions (2)

Part II: Parsing (fully bracketed binary) operator expressions

```
opExpParse
  = (token '(' >*>
     parser    >*>
     spot isOp >*>
     parser    >*>
     token ')')
     'build' makeExpr
```

Part III: Parsing literals (numerals)

```
litParse
  = ((optional (token '~')) >*>
     (neList (spot isDigit))
    'build' (charlistToExpr . uncurry (++))
```

A Parser for Expressions (3)

Two further parsers

```
neList   :: Parse a b -> Parse a [b]
optional :: Parse a b -> Parse a [b]
```

such that:

- `neList p` recognizes a non-empty list of the objects which are recognized by `p`.
- `optional p` recognizes an object recognized by `p` or succeeds immediately.

Note that `neList` and `optional` as well as a number of other supporting functions used such as:

- `isOp`
- `charlistToExpr`
- ...

are yet to be defined (⇝ homework).

The Top-level Parser: Putting it all Together

Converting a string to the expression it represents:

```
topLevel :: Parse a b -> [a] -> b
topLevel p inp
  = case results of
      [] -> error "parse unsuccessful"
      _  -> head results
    where
    results = [ found | (found, []) <- p inp ]
```

It is worth noting:

- The input string is provided by the value of `inp`.
- The parse is successful, if the result contains at least one parse, in which all the input has been read.

Summing up (1)

Parsers of the form:

```
type Parse a b = [a] -> [(b,[a])]

none :: Parse a b
succeed :: b -> Parse a b
spot :: (a -> Bool) -> Parse a a
alt :: Parse a b -> Parse a b -> Parse a b
>*> :: Parse a b -> Parse a c -> Parse a (b,c)
build :: Parse a b -> (b -> c) -> Parse a c
topLevel :: Parse a b -> [a] -> b
```

...support particularly well the construction of so-called recursive descent parsers.

Summing up (2)

The following language features proved invaluable for combinator parsing:

- Higher-order functions: `Parse a b` is of a functional type; all parser combinators are thus higher-order functions, too.
- Polymorphism: Consider again the type of `Parse a b`: We do need to be specific about either the input or the output type of the parsers we build. Hence, the above parser combinator can immediately be reused for other (token-) and data types.
- Lazy evaluation: "On demand" generation of the possible parses, automatical backtracking (the parsers will backtrack through the different options until a successful one is found).

Chapter 13.1: Further Reading (1)

- 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 13.1: 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 13.1: 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 13.1: 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 13.1: 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 13.2

Monadic Parsing

Monadic Parsing

The class Monad

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The type of a parser function:

```
newtype Parser a = Parser (String -> [(a,String)])
```

We then use the same convention as in Chapter 13.1, i.e.:

- Delivery of the empty list: Signals failure of the analysis.
- Delivery of a non-empty list: Signals success of the analysis; each element of the list is a pair, whose first component is the identified object (token) and whose second component the input still to be examined.

A Monad of Parsers

Basic Parsers:

- Recognizing single characters

```
item :: Parser Char
item  = Parser (\cs -> case cs of
                            ""     -> []
                            (c:cs) -> [(c,cs)])
```

Note:

- The functions `item` and `token` correspond to each other.

The Parser Monad (1)

Parser is a type constructor. This allows:

```
instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f
     = Parser (\cs -> concat [parse (f a) cs' |
                         (a,cs') <- parse p cs])
```

The Parser Monad (2)

Note:

- 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 (Parser p) = p`.
- The parser `p >>= f` first applies the parser `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 `f a` is a parser that is applied to the string `cs'`. The result is a list of lists that is then concatenated to give the final list of results.

The Parser Monad (3)

As required for instances of the class `Monad`, we can show that the 3 monad laws hold:

```
return a >>= f = f a
p >>= return = p
p >>= (\a -> (f a >>= g)) = (p >>= (\a -> f a)) >>= g
```

Properties of `return` and `(>>=)`

Reminder:

- The above properties are required for each instance of class `Monad`, not just for the specific instance of the parser monad
  - `return` is left-unit and right-unit for `(>>=)`
    $\leadsto$ allows a simpler and more concise definition of some parsers.
  - `(>>=)` is associative
    $\leadsto$ allows suppression of parentheses when parsers are applied sequentially.

The Parser Monad

This way we get another two important parsers:

- The always successful parser: `return`
- Sequencing of parsers: `(>>=)`

Note:

The functions

- `return` and `succeed`
- `(>>=)` and `(>*>)`

correspond to each other.

Typical Structure of a Parser (1)

Using the sequencing operator (>>=):

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

Typical Structure of a Parser (2)

Intuition:

There is a natural operational reading of such a parser:

- Apply parser `p1` and denote its result value `a1`.
- Apply subsequently parser `p2` and denote its result value `a2`.
- ...
- Apply concludingly parser `pn` and denote its result value `an`.
- Combine finally the intermediate result values by applying some suitable function `f`.

Typical Structure of a Parser (3)

The do-notation allows a more appealing notation:

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

Alternatively, in just one line:

```
do {a1 <- p1; a2 <- p2;...; an <- pn; f a1 a2...an}
```

Notational Conventions

Expressions of the form

- `ai <- pi` are called generators
  (since they generate values for the variables `ai`)

Remark:

A generator of the form `ai <- pi` can be

- replaced by `pi`, if the generated value will not be used afterwards.

Example: A Simple Parser

Write a parser p that

- reads three characters,
- drops the second character of these, and
- returns the first and the third character as a pair.

Implementation:

```
p :: Parser (Char,Char)
p  = do {c <- item; item; d <- item; return (c,d)}
```

Parser Extensions (1)

Monads with a `zero` and a `plus` are captured by two built-in class definitions in Haskell:

```
class Monad m => MonadZero m where
  zero :: m a

class MonadZero m => MonadPlus m where
  (++) :: m a -> m a -> m a
```

Parser Extensions (2)

The type constructor `Parser` can be made instances of these two classes as follows giving two further parsers:

- The parser that always fails:

```
instance MonadZero Parser where
  zero   = Parser (\cs -> [])
```

- The parser that non-deterministically selects:

```
instance MonadPlus Parser where
  p ++ q = (\cs -> parse p cs ++ parse q cs)
```

Simple Properties (1)

We can prove:

```
      zero ++ p  =  p
      p ++ zero  =  p
p ++ (q ++ r)  =  (p ++ q) ++ r
```

Informally:

- `zero` is left-unit and right-unit for `(++)`
- `(++)` is associative

Remark: The above properties are required to hold for each monad with `zero` and `plus`.

Simple Properties (2)

Specifically for the parser monad we can additionally prove:

```
               zero >>= f   =  zero
         p >>= const zero   =  zero
           (p ++ q) >>= f   =  (p >>= f) ++ (q >>= f)
p >>= (\a -> f a ++ g a)    =  (p >>= f) ++ (p >>= g)
```

Informally:

- `zero` is left-zero and right-zero element for `(>>=)`
- `(>>=)` distributes through `(++)`

Deterministic Selection

- The parser that deterministically selects:

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q
   = Parser (\cs -> case parse (p ++ q) cs of
                        []     -> []
                        (x:xs) -> [x])
```

It is worth noting:

- (+++) shows the same behavior as (++), but yields at most one result
- (+++) satisfies all of the previously listed properties of (++)

Further Parsers

Recognizing

- Single objects

```
char   :: Char -> Parser Char
char c = sat (c ==)
```

- Single objects satisfying a particular property

```
sat  :: (Char -> Bool) -> Parser Char
sat p
 = do {c <- item; if p c then return c else zero}
```

- Sequences of numbers, lower case and upper case characters, etc.
  ...analogously to `char`

It is worth noting:

- `sat` and `char` correspond to `spot` and `token`.

Recursion Combinators (1)

Useful parsers can often recursively be defined:

- Parse a specific string

```
string :: String -> Parser String
string "" = return ""
string (c:cs)
  = do {char c; string cs; return (c:cs)}
```

- Repeated applications of a parser p

  (Zero or more applications of p)

```
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)
```

Recursion Combinators (2)

- A variant of the parser many with interspersed applications of the 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)
```

Recursion Combinators (3)

- Repeated applications of a parser p, separated by applications of a parser op, whose result value is an operator that is assumed to associate to the left, and which is used to combine the results from the p parsers

```
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
```

Lexical Combinators (1)

Suitable combinators allow suppression of a lexical analysis (token recognition), which traditionally precedes parsing:

- Parsing of a string with blanks and line breaks

```
space :: Parser String
space  = many (sat isSpace)
```

- Parsing of a token by means of parsers p

```
token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}
```

Lexical Combinators (2)

- Parsing of a symbol token

```
symb    :: String -> Parser String
symb cs = token (string cs)
```

- Application of parser p, removal of initial blanks

```
apply   :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p})
```

Example: Parsing of Expressions (1)

Grammar:

...for arithmetic expressions built up from single digits using the operators +, -, *, /, and parentheses:

```
expr   ::=  expr addop term | term
term   ::=  term mulop factor | factor
factor ::=  digit | (expr)
digit  ::=  0 | 1 | ... | 9

addop  ::=  + | -
mulop  ::=  * | /
```

Example: Parsing of Expressions (2)

Parsing and evaluating expressions (yielding integer values) using the `chainl1` combinator to implement the left-recursive production rules for `expr` and `term`:

```
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: Parsing of Expressions (3)

Example:

Evaluating

```
apply expr " 1 - 2 * 3 + 4 "
```

gives the singleton list

```
[(-1,"")] as desired
```

as desired.

Chapter 13.2: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer Verlag, 2011. (Kapitel 19.10.5, 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 13.2: Further Reading (2)

- 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 13.2: Further Reading (3)

- 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

Logical Programming Functionally

Logical Programming Functionally

Declarative programming

- Functional style
- Logical style

If each of these two styles is appealing

- a combination of (features of) functional and logical programming

should be even more appealing.

Recent Article

- Sergio Antoy, Michael Hanus. Functional Logic Programming. Communications of the ACM 53(4):74-85, 2010.

...highlights the benefits of combining the paradigm features of both logical and functional programming.

Some of the essence of this article is summarized on the next couple of slides.

Evolution of Programming Languages

...the stepwise introduction of abstractions hiding the underlying computer hardware and the details of program execution.

- Assembly languages introduce mnemonic instructions and symbolic labels for hiding machine codes and addresses.
- FORTRAN introduces arrays and expressions in standard mathematical notation for hiding registers.
- ALGOL-like languages introduce structured statements for hiding gotos and jump labels.
- Object-oriented languages introduce visibility levels and encapsulation for hiding the representation of data and the management of memory.

Evolution of Prog. Lang. (Cont'd)

- Declarative languages, most prominently functional and logic languages hide the order of evaluation by removing assignment and other control statements.
  - A declarative program is a set of logical statements describing properties of the application domain.
  - The execution of a declarative program is the computation of the value(s) of an expression wrt these properties.

This way:

- The programming effort in a declarative language shifts from encoding the steps for computing a result to structuring the application data and the relationships between the application components.
- Declarative languages are similar to formal specification languages but executable.

Functional vs. Logic Languages

Functional languages

- are based on the notion of mathematical function
- programs are sets of functions that operate on data structures and are defined by equations using case distinction and recursion
- provide efficient, demand-driven evaluation strategies that support infinite structures

Logic languages

- are based on predicate logic
- programs are sets of predicates defined by restricted forms of logic foumulas, such as Horn clauses (implications)
- provide non-determinism and predicates with multiple input/output modes that offer code reuse

Functional Logic Languages: Examples (1)

- Curry

  Michael Hanus (Ed.). Curry: An Integrated Functional Logic Language (vers. 0.8.2, 2006).
  http://www.curry-language.org/
  (vers. 0.8.3, September 11, 2012),
  http://www.informatik.uni-kiel.de/∼curry/report.html

- 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.

Functional Logic Languages: Examples (2)

- 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`

Functional Logic Languages: Examples

And there are many more:

- Escher
- Oz
- HAL
- ...

A Curry Appetizer

Regular Expressions

```
data RE a = Lit a
          | Alt (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)
```

The Semantics of Regular Expressions

```
sem :: RE a -> [a]
sem (Lit c)    = [c]
sem (Alt r s)  = sem r ? sem s
sem (Conc r s) = sem r ++ sem s
sem (Star r)   = [] ? sem (Conc r (Star r))
```

Note: The Curry-operator ? denotes nondeterministic choice.

A Curry Appetizer (Cont'd)

```
abstar = Conc (Lit 'a') (Star (Lit 'b'))

sem abstar ->> ["a","ab","abb"]
```

The Curry-operator `=:=` indicates that an equation is to be solved rather than an operation to be defined; here it checks whether a given word `w` is in the language of a given regular expression `re`:

```
sem re =:= w
```

The following equation checks whether a string `s` contains a word generated by a regular expression `re` (similar to Unix's grep utility):

```
xs ++ sem re ++ ys =:= s
    where xs, ys free
```

In this chapter

...we will follow a different approach that has been presented in

- Michael Spivey, Silvija Seres. Combinators for Logic Programming. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

We will show how to

- integrate features of logical programming into functional programming.

Central means:

- Monads and monadic programming.

Declarative Programming

- Distinguishing: Emphasizes the "what", rather than the "how"
  - Essence: Programs are declarative assertions about a problem rather than imperative solution procedures.
- Variants: functional and logical programming.
- Question: Can functional and logical programming be uniformly combined?

Combining Features of Functional and Logical Programming

Basic approaches:

- Classical: Designing new programming languages, which enjoy features of both programming styles (e.g. Curry).
- Simpler: Implementing an interpreter for one style using the other style.
- Still simpler: Write "logical" programs in Haskell using a library of combinators.

  $\rightsquigarrow$ this is the approach taken in the following!

Further Reading

...on functional/logical programming languages:

- 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, 1995, 95-107.
- 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.

Remarks on the present Combinator Approach

Advantages and disadvantages

- compared to dedicated functional/logical programming languages
  - less costly
  - but less expressive

Key problems

- Modelling logical programs
  - yielding multiple answers
  - with logical variables (no distinction between input and output variables)
- Modelling the evaluation strategy inherent to logical programs

Running Example: Factoring of Nat. Numbers

Factoring of Natural Numbers: Decomposing a positive integer into the set of pairs of its factors.

Example:

Integer	Factor-Pairs
24	(1,24), (2,12), (3,8), (4,6), ..., (24,1)

Obvious Solution:

```
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)]
```

It is worth noting

The previous solution exploits:

- Explicit domain knowledge
  - E.g. $r * s = n \;\Rightarrow\; r \leq n \wedge s \leq n$
  - This renders possible: Restriction to a finite search space `[1..24]`$\times$`[1..24]`

Often such knowledge is not available; in general:

- The search space cannot be restricted a priori
- In the following thus: Considering the factoring problem as a search problem over an infinite search space `[1..]`$\times$`[1..]`

Tackling the 1st Problem: Several Results

Solution: Lists of successes

⇝lazy lists (Phil Wadler)

Idea

- A 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 (list of successes) are provided as they are found, rather than as a complete list after termination of the computation.

Back to the Example

Realizing this idea in the factoring example (assuming that the search space cannot be bounded a priori):

```
factor   :: Int -> [(Int,Int)]
factor n =  [(r,s) | r<-[1..], s<-[1..], r*s == n]

?factor 24
[(1,24)
```

...followed by an infinite wait.

$\rightsquigarrow$ This is of no practical value!

Remedy: Fair Order via Diagonalization (1)

Explore the search space of pairs in a fair order:

```
factor n
  = [(r,s) | (r,s)<-diagprod [1..][1..], r*s == n]
```

where

```
diagprod        :: [a] -> [b] -> [(a,b)]
diagprod xs ys
  = [(xs!!i, y!!(n-i)) | n<-[0..], i<-[0..n]]
```

Effect: Each pair (x,y) is now reached after a finite number of steps:

```
[(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),
(2,3),(3,2),...]
```

Remedy: Fair Order via Diagonalization (2)

Applied to our running example, we obtain:

```
?factor 24
[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)
```

...this means, we obtain all results; followed again, however, by an infinite wait.

Of course, this was expected, since the search space is infinite.

Systematic Remedy: Using Monads

Reminder:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Notational conventions for the following development:

- `Stream a` ...for (potentially) infinite lists
- `[a]` ...for finite lists
- Note: The distinction between `Stream a` for infinite lists and `[a]` for finite lists is only conceptually; the following definition makes this explicit:

  ```
  type Stream a = [a]
  ```

List Monad

The monad of (potentially infinite) lists

Definition of the monad operations

- `return` (yields the singleton list):

```
return :: a -> Stream a
return x = [x]
```

- binding operator (>>=):

```
(>>=) :: Stream a -> (a -> Stream b) -> Stream b
xs >>= f = concat (map f xs)
```

Other monad operations are irrelevant in our context.

Benefit

The monad operations `return` and `(>>=)` allow us to model/replace list comprehension:

The meaning of the expression, for example,

```
[(x,y) | x <- [1..], y <- [10..]]
```

...using list comprehension is equivalent to

```
concat (map (\x -> [(x,y) | y <- [10..]])[1..])
```

...that itself is equivalent to

```
concat (map (\x ->
         concat (map (\y -> [(x,y)])[10..]))[1..])
```

Using `return` and `(>>=)` this can concisely be expressed by:

```
[1..] >>= (\x -> [10..] >>= (\y -> return (x,y)))
```

Benefit (Cont'd)

Haskell's do-notation allows an even more compact equivalent representation:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

Recalling the general rule:

The expression

```
do x1 <- e1; x2 <- e2; ... ; xn <- en; e
```

is a shorthand for

```
e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```

Fairness: Adapting the binding op. (>>=) (1)

Are we done? Not yet, since:

- Exploring the pairs of the search space is still not fair.

The expression

```
do x <- [1..]; y <- [10..]; return (x,y)
```

yields the stream

```
[(1,10),(1,11),(1,12),(1,13),(1,14),..
```

This problem is going to be tackled next.

Fairness: Adapting the binding op. (>>=) (2)

Idea: Embedding diagonalization into (>>=)

Implementation

Introducing a new type `Diag a`:

```
newtype Diag a = MkDiag (Stream a) deriving Show
```

...together with an auxiliary function for stripping off the type constructor `MkDiag`:

```
unDiag (MkDiag xs) = xs
```

Fairness: Adapting the binding op. (>>=) (3)

Diag is made an instance of the 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)
```

Fairness: Adapting the binding op. (>>=) (4)

where

```
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)
```

It is worth noting:

- `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.

Fairness: Adapting the binding op. (>>=) (5)

Intuition:

- `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).
- `lzw` reminds to "like zipWith."

Fairness: Adapting the binding op. (>>=) (6)

The idea underlying `diag`:

- Transform an infinite list of infinite lists
  `[[x11,x12,x13,..],[x21,x22,..],[x31,x32,..],..]`
- ...into an infinite list of finite diagonals
  `[[x11],[x12,x21],[x13,x22,x31],..]`

This way:

```
?do x <- MkDiag [1..]; y<-MkDiag [10..]; return (x,y)
MkDiag[(1,10),(1,11),(2,10),(1,12),(2,11),
       (3,10),(1,13),..
```

Summing up

- We have achieved: The pairs are delivered in a fair order!

Back to the Factoring Problem (1)

Current state of our solution:

- Generating pairs (in a fair order): done.
- Selecting (those pairs being part of the solution): still open.

Approach for solving the selection problem, i.e., filtering out the pairs (r, s) satisfying the equality $r \times s = n$:

- Filtering with conditions!

Back to the Factoring Problem (2)

For that purpose:

```
class Monad m => Bunch m where
  zero :: m a           -- empty result, no answer
  alt  :: m a -> m a -> m a -- all answers either
                            -- in xm or ym
  wrap :: m a -> m a    -- answers yielded by auxi-
                        -- liary calculations; right
                        -- now, wrap is defined as
                        -- the identity function
```

It is worth noting:

- The value `zero` allows to express an empty answer set.

Back to the Factoring Problem (3)

In detail:

The instance declaration for ordinary lazy lists:

```
instance Bunch [] where
 zero       = []
 alt xs ys = xs ++ ys
 wrap xs    = xs
```

...and for the monad Diag:

```
instance Bunch Diag where
 zero                               = MkDiag[]
 alt (MkDiag xs)(MkDiag ys) = MkDiag (shuffle xs ys)
 wrap xm                            = xm

shuffle [] ys      = ys
shuffle (x:xs) ys = x : shuffle ys xs
```

(Remark: alt and wrap will be used later.)

Back to the Factoring Problem (4)

By means of zero, the function test yields the key for filtering:

```
test   :: Bunch m => Bool -> m()
test b =  if b then return() else zero
```

This does not look useful, but it provides the key to filtering:

```
?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,..
```

Are we done? (1)

Not yet!

Consider:

```
?do r <- MkDiag[1..]; s <- MkDiag[1..];
    test(r*s==24); return (r,s)
MkDiag[(1,24)
```

...followed by an infinite wait.

What are the reasons for that?

```
do r <- MkDiag[1..]; s <- MkDiag[1..];
   test(r*s==24); return (r,s)
```

is equivalent to

```
do x <- MkDiag[1..]
   (do y <- MkDiag[1..]; test(x*y==24);
       return (x,y))
```

Are we done? (2)

I.e., the generator for y is merged with the subsequent test to the following (sub-) expression:

```
do y <- MkDiag[1..]; test(x*y==24); return (x,y)
```

Intuition:

- 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 search in vain for further values of y.
- For $x = 5$ we thus do not observe any output.

Solution Approach

The deeper reason for this undesired behaviour:

The missing associativity of `(>>=)` for `Diag`, i.e.,

```
(xm >>= f) >>= g  =  xm >>= (\x -> f x >>= g)
```

...does not hold for `(>>=)` and `Diag!`

Remedy: Explicit grouping of generators to ensure fairness

```
?do (x,y) <- (do u <- MkDiag[1..];
    v <- MkDiag[1..]; return (u,v))
    test (x*y==24); return (x,y)
MkDiag[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),
       (1,24),(24,1)
```

...all results, subsequently followed by an infinite wait.

Remarks

- All results, subsequently followed by an infinite wait

  ⇝ this is the best we can hope for if the search space is infinite.
- Explicit grouping

  ⇝ required only because of missing associativity of `(>>=)`, otherwise both expressions would be equivalent.
- In the following

  ⇝ avoid infinite waiting by indicating that a result has not (yet) been found.

Indicating that no solution is found

To this purpose: Introducing a new type `Matrix` together with breadth search.

Intuition

- Type `Matrix`: Infinite list of finite lists.
- Goal: A program that yields a matrix of answers, where row i contains all answers that can be computed with costs $c(i)$.
- Solving the indication problem: By returning the empty list in a row (means "nothing found").

Implementation (1)

The new type `Matrix`

```
newtype Matrix a
  = MkMatrix (Stream [a]) deriving Show
```

...with an auxiliary function for stripping off the constructor:

```
unMatrix (MkMatrix xm) = xm
```

Implementation (2)

Preliminary definitions for making Matrix an instance of class Bunch:

```
return x = MkMatrix[[x]]  -- Matrix with a single row
zero = MkMatrix[]          -- Matrix without rows
alt(MkMatrix xm) (MkMatrix ym) = MkMatrix(lzw (++) xm ym)
wrap(MkMatrix xm) = MkMatrix([]:xm)  -- the clou is en-
                                      -- coded in wrap!

(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))

bindm :: Stream[a] -> (a -> Stream[b]) -> Stream[b]
bindm xm f = map concat (diag (map (concatAll . map f) xm))

concatAll :: [Stream [b]] -> Stream [b]
concatAll = foldr (lzw (++)) []
```

Implementation (3)

In total we are now ready to make Matrix an instance of the 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)                = MkMatrix([]:xm)

intMat = MkMatrix[[n] | n <- [1..]]
```

Example:

```
?do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
MkMatrix[[],[],[],[],[],[],[],[],[(4,6),(6,4)],
  [(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],
  [[],[],[],[],[],[],[],(1,24),(24,1)],[],[],[],..
```

A Variety of Search Strategies

(i) Breadth search (`MkMatrix[[n]|n<-[1..]]`), (ii) depth search (`[1..]`), (iii) diagonalization:

...by means of additional functions that allow us to fix the strategy of interest at the time of calling ("just in time").

Control via a monad type:

```
choose          :: Bunch m => Stream a -> m a
choose (x:xs) =  wrap (return x 'alt' choose xs)

factor   :: Bunch m => Int -> m(Int, Int)
factor n =  do r <- choose[1..]; s <- choose[1..];
               test(r*s==n); return (r,s)
```

Selecting a Search Strategy

This allows:

- Usage of `factor` with different search strategies.
- The specified type of `factor` determines the search monad (and thus the search strategy).

```
?factor 24 :: Stream(Int,Int)
[(1,24)

?factor 24 :: Matrix(Int, Int)
Matrix[[],[],[],[],[],[],[],[],[],[],[(4,6),(6,4)],
  [(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],[],
  [],[],[],[],[],[],[(1,24),(24,1)],[],[],[],..
```

Summary of Progress

Recall:

The 3 key problems we had/have to deal with:

- Modelling logical programs with
  - multiple results: done (essentially by means of lazy lists)
  - logical variables: still open
    - Common for logical programs: not a pure simplification of an initially completely given expression, but a simplification of an expression containing variables, for which appropriate values have to be determined. In the course of the computation, variables can be replaced by other subexpressions containing variables themselves, for which then appropriate values have to be found.
  - Modelling of the evaluation strategy inherent to logical programs: done
    - implicit search of logical programming languages has been made explicit.
    - by means of type classes of Haskell even different search strategies were conveniently be realizable.

Tackling the Final Problem: Terms, Substitutions & Predicates (1)

Towards the modeling in Haskell:

Terms will describe values of logical variables:

```
data Term = Int Int
          | Nil
          | Cons Term Term
          | Var Variable deriving Eq
```

Named variables will be used for formulating queries, generated variables evolve in the course of the computation:

```
data Variable = Named String
              | Generated Int deriving (Show, Eq)
```

Terms, Substitutions & Predicates (2)

Some auxiliary functions

- for transforming a string into a named variable

```
var   :: String -> Term
var s =  Var (Named s)
```

- for constructing a term representation of a list of integers

```
list    :: [Int] -> Term
list xs =  foldr Cons Nil (map Int xs)
```

Terms, Substitutions & Predicates (3)

Substitution and unification:

```
-- Substitution is essentially a mapping
-- from variables to terms (details later)
newtype Subst
```

Further support functions:

```
apply :: Subst -> Term -> Term
idsubst :: Subst
unify :: (Term, Term) -> Subst -> Maybe Subst
```

Terms, Substitutions & Predicates (4)

Logical programs (in our Haskell environment) with `m` of type `bunch`:

```
-- Logical programs have type Pred m
type Pred m = Answer -> m Answer

-- Answers; the integer-component controls
-- the generation of new variables
newtype Answer = MkAnswer (Subst, Int)
```

Terms, Substitutions & Predicates (5)

```
-- "Initial answer"
initial :: Answer
initial =  MkAnswer (idsubst, 0)
run    :: Bunch m => Pred m -> m Answer
run p = p initial

-- "Program run of a predicate as query", where
-- p is applied to the initial answer
run p :: Stream Answer
```

Writing logical programs

Example:

append(a,b,c) where a,b denote lists and c the concatenation of the lists a and b.

Implementation as a function of terms on predicates:

```
append :: Bunch m => (Term, Term, Term) -> Pred m

-- The implementation of append (will follow!) and
-- of appropriate Show-Functions is supposed:
?run(append(list[1,2],list[3,4],var "z"))
                                     :: Stream Answer
[z=[1,2,3,4]]

-- Note: Equivalent to the above list but more
-- accurate would be:
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

Combinators for logical programs (1)

Simple predicates are formed by means of the operators (=:=) (equality of terms):

```
?run(var "x" =:= Int 3) :: Stream Answer
[{x=3}]
```

Implementation of (=:=) by means of `unify`:

```
(=:=) :: Bunch m => Term -> Term -> Pred m
(t=:=u)(MkAnswer(s,n)) =
   case unify(tu) s of
     Just s' -> return(MkAnswer(s',n))
     Nothing -> zero
```

Combinators for logical programs (2)

Conjunction of predicates by means of the operator (&&&) (conjunction):

```
?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 by means of the operator (>>=) of type bunch:

```
(&&&) :: Bunch m => Pred m -> Pred m -> Pred m
(p &&& q) s = p s >>= q
-- equivalent and highlighting the
-- sequentiality would be
do t <- p s; u <- q t; return u
```

Combinators for logical programs (3)

Disjunction of predicates by means of the operator (|||) (Disjunction):

```
?run(var "x" =:= Int 3 ||| var "x" =:= Int 4)
                                        :: Stream Answer
[{x=3,x=4}]
```

Implementation by means of the operator alt of type bunch:

```
(|||) :: Bunch m => Pred m -> Pred m -> Pred m
(p ||| q) s = alt (p s) (q s)
```

Combinators for logical programs (4)

Introducing new variables in predicates (exploiting the integer-component of answers)
...on the construction of local variables in recursive predicates:

```
exists :: Bunch m => (Term -> Pred m) -> Pred m
exists p (MkAnswer (s,n)) =
  p (Var(Generated n)) (MkAnswer(s,n+1))
```

Also for handling recursive predicates
...ensures that in connection with `Matrix` the costs per recursion unfolding increase by 1:

```
step       :: Bunch m => Pred m -> Pred m
step p s =  wrap (p s)
```

Example

Examples of applications of wrap and step:

```
?run (var "x" =:= Int 0) :: Matrix Answer
MkMatrix[[{x=0}]]

?run(step(var "x" =:= Int 0)) :: Matrix Answer
MkMatrix[[],[{x=0}]]
```

Recursive Programs (1)

This allows us to provide the implementation of append:

```
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)))))
```

Recursive Programs (2)

As common for logical programs, also the following application of append is possible:

- The concatenation of which lists equals the list [1,2,3]?

```
?run(append(var "x", var "y", list[1,2,3]))
                                    :: Stream Answer
[{x = Nil, y = [1,2,3]},
  {x = [1], y = [2,3]},
  {x = [1,2], y = [3]},
  {x = [1,2,3], y = Nil}]
```

A More Complex Example (1)

Constructing "good" sequences consisting of 0s and 1s.

Definition:

1. The sequence `[0]` is good.
2. If the sequences `s1` and `s2` are good, then also the sequence `[1] ++ s1 ++ s2`.
3. Except of the sequences according to 1. and 2., there are no other good sequences.

A More Complex Example (2)

Implementation as a predicate:

```
good(s) =
  step (s =:= Cons(Int 0) Nil
    ||| exist (\t -> exists (\q -> exists (\r ->
            s =:= Cons (Int 1) t &&& append(q,r,t)
            &&& good(q) &&& good(r)))))
```

Applications (1)

1) Test of being "good":

```
?run (good (list[1,0,1,1,0,0,1,0,0]))
                                    :: Stream Answer
[{}] -- empty answer set, if argument list is good

?run (good (list[1,0,1,1,0,0,1,0,1]))
                                    :: Stream Answer
[]   -- no answer, if argument list is not good
```

It is worth noting:

- "empty answer" and "no answer" correspond to "yes" and "no" of a Prolog system.

Applications (2)

2) Constructing "good" lists

```
-- With an unfair bunch-type: Some answers
-- are missing
?run(good(var "s")) :: Stream Answer
[s=[0],
  s=[1,0,0],
  s=[1,0,1,0,0],
  s=[1,0,1,0,1,0,0],
  s=[1,0,1,0,1,0,1,0,0],..
```

Applications (3)

```
-- For comparison: With a fair bunch-type
?run(good(var "s")) :: Diag Answer
Diag[s=[0],
  s=[1,0,0],
  s=[1,0,1,0,0],
  s=[1,0,1,0,1,0,0],
  s=[1,1,0,0,0],
  s=[1,0,1,0,1,0,1,0,0],
  s=[1,1,0,0,1,0,0],
  s=[1,0,1,1,0,0,0],
  s=[1,1,0,0,1,0,1,0,0],..
```

Applications (4)

```
-- For comparison: With a breadth-first search
-- bunch-type. Effect: The output of results is
-- more "predictable".
?run(good(var "s")) :: Matrix Answer
MkMatrix[[],
    [s=[0]],[],[],[],
    [s=[1,0,0]],[],[],[],
    [s=[1,0,1,0,0]],[],
    [s=[1,1,0,0,0]],[],
    [s=[1,0,1,0,1,0,0]],[],
    [s=[1,0,1,1,0,0,0]],s=[1,1,0,0,1,0,0]],[],
    ..
```

Delivering Missing Definitions (1)

Priorities of new infix operators:

```
infixr 4 =:=
infixr 3 &&&
infixr 2 |||
```

Substitution:

```
newtype Subst = MkSubst [(Var, Term)]
unSubst(MkSubst s) = s

idsubst = MkSubst[]
extend x t (MkSubst s) = MkSubst ((x,t):s)
```

Delivering Missing Definitions (2)

Application of substitution:

```
apply :: Subst -> Term -> Term
apply s t =
  case deref s t of
    Cons x xs -> Cons (apply s x) (apply s xs)
    t'        -> t'

deref :: Subst -> Term -> Term
deref s (Var v) =
  case lookup v (unSubst s) of
    Just t    -> deref s t
    Nothing   -> Var v
deref s t = t
```

Delivering Missing Definitions (3)

Unification:

```
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
```

Delivering Missing Definitions (4)

```
occurs :: Variable -> Term -> Subst -> Bool
occurs x t s =
  case deref s t of
    Var y       -> x == y
    Cons y ys -> occurs x y s || occurs x ys s
    _           -> False
```

Summing up

Current functional logic languages aim at balancing

- generality (in terms of paradigm integration)
- efficient implementations

Functional logic programming offers

- support of specification, prototyping, and application programming within a single language
- terse, yet clear, support for rapid development by avoiding some tedious tasks, and allowance of incremental refinements to improve efficiency

Overall: Functional logic programming

- an emerging paradigm with appealing features

Chapter 14: Further Reading (1)

- Hassan Ait-Kaci, Roger Nasr. *Integrating Logic and Functional Programming*. Lisp and Symbolic Computation 2(1):51-89, 1989.
- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
- 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 (2)

- 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 (3)

- 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.
- 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.
- John W. Lloyd. *Programming in an Integrated Functional and Logic Language*. Journal of Functional and Logic Programming 1999(3), 49 pages, MIT Press, 1999.

Chapter 14: Further Reading (4)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 22, Integration von Konzepten anderer Programmiersprachen)
- 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.

Chapter 14: Further Reading (5)

- Silvija Seres, Michael Spivey. *Embedding Prolog in Haskell*. In Proceedings of the 1999 Haskell Workshop (Haskell'99), 25-38, 1999.
- Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 177-199, 2003.

Chapter 15

Pretty Printing

Motivation

Pretty Printing is

- like lexical and syntactical analysis another typical application used for demonstrating the elegance of functional programming.

What's it all about?

A pretty-printer is

- a tool (often a library of routines) designed for converting a tree into plain text.

Essential goal of pretty printing:

- Preserving and reflecting the structure of the tree by indentation while using a minimum number of lines.

Hence

- Pretty printing can be considered the converse problem to parsing.

A “Good” Pretty-Printer

...is distinguished by properly balancing

- Simplicity of usage
- Flexibility of the format
- “Prettiness” of output

The presentation in this chapter

...is based on:

- Philip Wadler. A Prettier Printer. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

It shall improve (see end of chapter) on the below pretty printer library proposed by John Hughes that 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 Simple Pretty Printer: Basic Approach

Requirement: For each document there shall be only one possible layout (e.g., no attempt is made to compress structure onto a single line).

The basic operators needed are:

```
(<>)   :: Doc -> Doc -> Doc  -- associative concate-
                                nation of documents
nil    :: Doc                -- The empty document:
                                Right and left unit
                                for (<>)
text   :: String -> Doc      -- Conversion function:
                                Converts a string to
                                a document
line   :: Doc                -- Line break
nest   :: Int -> Doc -> Doc  -- Adding indentation
layout :: Doc -> String      -- Output: Converts a
                                document to a string
```

Convention

- Arguments of `text` are free of newline characters.

A Simple Implementation

Implement

- `doc` as strings (i.e. as data type `String`)

with

- `(<>)` as concatenation of strings
- `nil` as empty string
- `text` as identity on strings
- `line` as new line
- `nest i` as indentation: adding *i* spaces (after each line break by means of `line`) ⇝ essential difference to Hughes' pretty printer that also allows inserting spaces in front of strings allowing here to drop one concatenation operator
- `layout` as identity on strings

Example

Converting trees into documents (here: Strings) which are output as text (here: Strings).

Consider the following type of trees:

```
data Tree = Node String [Tree]
```

A concrete value B of type Tree:

```
Node "aaa" [Node "bbbbb" [Node "cc" [], Node "dd" []],
            Node "eee" [],
            Node "ffff" [Node "gg" [],
                         Node "hhh" [],
                         Node "ii" []
                        ]
           ]
```

...and its desired output

A text, where indentation reflects the structure of tree B:

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```

It is worth noting:

- Sibling trees start on a new line, properly indented.

Implementation

The below implementation achieves this:

```
data Tree              = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) = text s <>
                       nest (length s) (showBracket ts)

showBracket :: [Tree] -> Doc
showBracket []         = nil
showBracket ts         = text "[" <>
                       nest 1 (showTrees ts) <> text "]"

showTrees :: [Tree] -> Doc
showTrees [t]          = showTree t
showTrees (t:ts)       = showTree t <> text "," <>
                       line <> showTrees ts
```

Another possibly wanted output of B

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

Note:

- each subtree starts on a new line, properly indented.

An implementation producing the latter output

```
data Tree = Node String [Tree]

showTree’ :: Tree -> Doc
showTree’ (Node s ts) = text s <> showBracket’ ts

showBracket’ :: [Tree] -> Doc
showBracket’ [] = nil
showBracket’ ts = text "[" <> nest 2 (line <>
                  showTrees’ ts) <> line <> text "]"

showTrees’ :: [Tree] -> Doc
showTrees’ [t]    = showTree t
showTrees’ (t:ts) = showTree t <> text "," <> line
                                         <> showTrees ts
```

Normal Form of Documents

Documents can always be reduced to normal form.

Normal form

- Text alternating with line breaks nested to a given indentation:

  ```
  text s0 <> nest i1 line <> text s1 <> ...
                 <> nest ik line <> text sk
  ```

  where

  - each s_j is a (possibly empty) string
  - each i_j is a (possibly zero) natural number

Example on Normal Forms 1(2)

A document

```
text "bbbbb" <> text "[" <>
nest 2 (
     line <> text "ccc" <> text "," <>
     line <> text "dd"
) <>
line <> text "]"
```

...and how it is output:

```
bbbbb[
  ccc,
  dd
]
```

Example on Normal Forms 2(2)

The same document

```
text "bbbbb" <> text "[" <>
nest 2 (
     line <> text "ccc" <> text "," <>
     line <> text "dd"
) <>
line <> text "]"
```

...and its normal form:

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

Why does it work?

...because of the properties (laws) the functions enjoy.

In more detail

...because of the fact that

- `(<>)` is associative with unit `nil`
- the laws summarized on the next slide.

  Note:

  - All of these laws except of the last one are paired; they are paired with a corresponding law for their units.

Properties of the Functions/Laws 1(2)

We have the following (pairs of) laws (except for the last one):

```
text (s ++ t)    = text s <> text t -- text is a homomor-
text ""          = nil              -- phism from string
                                    -- concatenation to
                                    -- document concate-
                                    -- nation

nest (i+j) x     = nest i (nest j x) -- nest is a homomor-
nest 0 x         = x                 -- phism from addition
                                     -- to composition

nest i (x <> y) = nest i x <> nest i y -- nest distributes
nest i nil      = nil                  -- through document
                                       -- concatenation

nest i (text s) = text s -- Nesting is absorbed by text;
                         -- different to Hughes' pretty
                         -- printer)
```

Properties of the Functions/Laws 2(2)

Relevance and Impact

- The above laws are sufficient to ensure that documents can always be transformed into normal form
  - First four laws: applied from left to right
  - Last three laws: applied from right to left

Further Properties/Laws

...that relate documents to their layouts:

```
layout (x <> y)       = layout x ++ layout y
layout nil            = ""  -- layout is a homomorphism
                            -- from document concate-
                            -- nation to string conca-
                            -- tenation

layout (text s)       = s   -- layout is the inverse
                            -- of text

layout (nest i line) = '\n' : copy i ' '
                            -- layout of a nested line
                            -- is a newline followed by
                            -- one space for each level
                            -- of indentation
```

The Implementation of Doc

Intuition

- Represent documents as a concatenation of items, where each item is a text or a line break indented to a given amount.

This is realized as a sum type (the algebra of documents):

```
data Doc = Nil
         | String 'Text' Doc
         | Int 'Line' Doc
```

The constructors relate to the document operators as follows:

```
Nil          = nil
s 'Text' x = text s <> x
i 'Line' x = nest i line <> x
```

Example

Using this new algebraic type Doc, the normal form (considered previously)

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

...is represented by the following value of Doc:

```
"bbbbb[" 'Text' (
2 'Line' ("ccc," 'Text' (
2 'Line' ("dd," 'Text' (
0 'Line' ("]," 'Text' Nil)))))
```

Derived Implementations 1(2)

Implementations of the document operators can easily be derived from the above equations:

```
nil                =  Nil
text s             =  s 'Text' Nil
line               =  0 'Line' Nil

(s 'Text' x) <> y  =  s 'Text' (x <> y)
(i 'Line' x) <> y  =  i 'Line' (x <> y)
Nil <> y           =  y
```

Derived Implementations 2(2)

```
nest i (s 'Text' x) = s 'Text' nest i x
nest i (j 'Line' x) = (i+j) 'Line' nest i x
nest i Nil = Nil

layout (s 'Text' x) = s ++ layout x
layout (i 'Line' x) = '\n' : copy i ' ' ++ layout x
layout Nil          = ""
```

Correctness of the derived Implementations

...can be shown for each of them, e.g.:

- Derivation of
  `(s 'Text' x) <> y = s 'Text' (x <> y)`

```
  (s 'Text' x) <> y
=   { Definition of Text }
  (text s <> x) <> y
=   { Associativity of <> }
  text s <> (x <> y)
=   { Definition of Text }
  s 'Text' (x <> y)
```

The remaining equations can be shown using similar reasoning.

Documents with Multiple Layouts

Adding Flexibility:

- Up to now: Documents were equivalent to a string (i.e., they have a fixed single layout)
- Next: Documents shall be equivalent to a set of strings (i.e., they may have multiple layouts)

where each string corresponds to a layout.

This can be rendered possible by just adding a new function:

```
group :: Doc -> Doc
```

Informally:

Given a document, representing a set of layouts, `group` returns the set with one new element added that represents the layout in which everything is compressed on one line: Replace each newline (plus indentation) by a single space.

Preferred Layouts

"Beauty" needs to be specified/defined:

- `pretty` replaces `layout`

  ```
  pretty :: Int -> Doc -> String
  ```

  and picks the prettiest layout depending on the preferred maximum line width argument.

  Remark: `pretty`'s integer-argument specifies the preferred maximum line length of the output (and hence the prettiest layout out of the set of alternatives at hand).

Example

Using the modified showTree function based on group:

```
showTree (Node s ts)
  = group (text s
          <> nest (length s) (showBracket ts))
```

...the call of pretty 30 (once ompletely specified) will yield the output:

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

This ensures:

- Trees are fit onto one line where possible (i.e., length ≤ 30).
- Insertion of sufficiently many line breaks in order to avoid exceeding the given maximum line length.

Implementation of the new Functions

The following supporting functions are required:

```
-- Forming the union of two sets of layouts
(<|>)   :: Doc -> Doc -> Doc

-- Replacement of each line break (and its
-- associated indentation) by a single space
flatten :: Doc -> Doc
```

Implementation of the new Functions (Cont'd)

- Observation: A document always represents a non-empty set of layouts.
- Requirements:
  - In `(x <|> y)` all layouts of `x` and `y` enjoy the same flat layout (mandatory invariant of `<|>`).
  - Each first line in `x` is at least as long as each first line in `y` (second invariant).
- Note: `<|>` and `flatten` are not directly exposed to the user (only via `group` and other supporting functions).

Properties/Laws of (<|>)

Operators on simple documents are extended pointwise through union:

```
(x <|> y) <> z   = (x <> z) <|> (y <> z)
x <> (y <|> z)   = (x <> y) <|> (x <> z)
nest i (x <|> y) = nest i x <|> nest i y
```

Properties/Laws of `flatten`

The interaction of `flatten` with other document operators:

```
flatten (x <|> y) = flatten x -- distribution law

flatten (x <> y)  = flatten x <> flatten y
flatten nil       = nil
flatten (text s)  = text s
flatten line      = text " " -- the most intere-
                             -- sting case: line-
                             -- breaks are replaced
                             -- by a single space
flatten (nest i x) = flatten x
```

Implementation of group

...by means of `flatten` and `(<>)`, the implementation of `group` can be given:

```
group x = flatten x <|> x
```

Intuitively: `group` adds the flattened layout to a set of layouts.

Note: A document always represents a non-empty set of layouts where all layouts in the set flatten to the same layout.

Normal Form

Based on the previous laws each document can be reduced to a normal form of the form

```
x1 <|> ... <|> xn
```

where each `xi` is in the normal form of simple documents (which was introduced previously).

Selecting a “best” Layout out of a Set of Layouts

...by defining an ordering relation on lines in dependence of the given maximum line length.

Out of two lines

- which do not exceed the maximum length, select the longer one
- of which at least one exceeds the maximum length, select the shorter one

Note: Sometimes we have to pick a layout where some line exceeds the limit (a key difference to the approach of Hughes). However, this is done only, if this is unavoidable.

The Adapted Implementation of Doc

The new implementation of Doc as algebraic type. It is similar to the previous one except for the new construct representing the union of two documents:

```
data Doc = -- As before: The first 3 alternatives
           Nil
         | String 'Text' Doc
         | Int 'Line' Doc
           -- New: We add a construct representing
              the union of two documents
         | Doc 'Union' Doc
```

Relationship of Constructors and Document Operators

The following relationships hold between the constructors and the document operators:

```
Nil            = nil
s 'Text' x     = text s <> x
i 'Line' x     = nest i line <> x
x 'Union' y = x <|> y
```

Example 1(8)

The document

```
group(
   group(
        group(
           group( text "hello" <> line <> text "a")
        <> line <> text "b")
    <> line <> text "c")
<> line <> text "d")
```

Example 2(8)

...has the following 5 possible layouts:

```
hello a b c d    hello a b c    hello a b    hello a    hello
                 d              c            b          a
                                d            c          b
                                             d          c
                                                        d
```

Example 3(8)

Task: Print the above document under the constraint that the maximum line width is 5.

⇝ the right-most layout of the previous slide is requested.

Initial (performance) considerations:

- Factoring out "hello" of all the layouts in x and y

```
"hello" 'Text' ((" " 'Text' x) 'Union' (0 'Line' y))
```

- Defining additionally the interplay of (<>) and nest with Union

```
(x 'Union' y) <> z   = (x <> z) 'Union' (y <> z)
nest k (x 'Union' y) = nest k x 'Union' nest k y
```

Example 4(8)

Implementations of group and flatten can easily be 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
```

Example 5(8)

Considerations on correctness (similar reasoning as earlier):

Derivation of `group (i 'Line' x)` (see line two) (preserving the invariant required by `union`)

```
  group (i 'Line' x)
=   { Definition of Line }
  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 }
  (" " 'Text' flatten x) 'Union' (i 'Line' x)
```

Example 6(8)

Correctness considerations (cont'd):

Derivation of `group (s 'Text' x)` (see line three)

```
  group (s 'Text' x)
=   { Definition Text }
  group (text s <> x)
=   { Definition group }
  flatten (text s <> x) <|> (text s <> x)
=   { Definition flatten }
  (text s <> flatten x) <|> (text s <> x)
=   { <> distributes through <|> }
  text s <> (flatten x <|> x)
=   { Definition group }
  text s <> group x
=   { Definition Text }
  s 'Text' group x
```

Example 7(8)

Selecting the "best" layout:

```
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
```

Remark:

- `best`: Converts a "union"-afflicted document into a "union"-free document.
- Argument `w`: Maximum line width.
- Argument `k`: Already consumed letters (including indentation) on current line.

Example 8(8)

Check, if the first document line stays within the maximum line length w:

```
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 (layout remains unchanged):

Select the best layout and convert it to a string:

```
pretty w x = layout (best w 0 x)
```

Enhancing Performance: A More Efficient Variant

Sources of inefficiency:

1. Concatenation of documents might pile up to the left.
2. Nesting of documents adds a layer of processing to increment the indentation of the inner document.

Problem fix:

- For 1.): Add an explicit representation for concatenation, and generalize each operation to act on a list of concatenated documents.
- For 2.): Add an explicit representation for nesting, and maintain a current indentation that is incremented as nesting operators are processed.

Enhancing Performance: A More Efficient Variant (Cont'd)

Implementing this fix by means of a new implementation of documents:

```
data DOC = NIL              -- Here is one constructor
         | DOC :<> DOC      -- corresponding to each
         | NEST Int DOC     -- operator that builds a
         | TEXT String      -- document
         | LINE
         | DOC :<|> DOC
```

Remark:

- In distinction to the previous document type we here use capital letters in order to avoid name clashes with the previous definitions

Implementing the Document Operators

Defining the operators to build a document are straightforward:

```
nil      = NIL
x <> y   = x :<> y
nest i x = NEST i x
text s   = TEXT s
line     = LINE
```

Implementing group and flatten

As before, we require the following invariants:

- In (x :<|> y) all layouts in x and y flatten to the same layout.
- No first line in x is shorter than any first line in y.

Definitions of group and flatten are then straightforward:

```
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
```

Representation Function

Generating the document from an indentation-afflicted document ("indentation-document pair"):

```
rep z = fold (<>) nil [nest i x | (i,x) <- z ]
```

Selecting the "best" Layout

Generalizing the function "best" by composing the old function with the representation function to work on lists of indentation-document pairs:

```
be w k z = best w k (rep z)        (Hypothesis)

best w k x                  = be w k [(0,x)]
```

where the definition is derived from the old one:

```
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))
```

Preparing the XML-Application 1(3)

First some useful supporting functions:

```
x <+> y            = x <> text " " <> y
x </> y            = x <> line <> y

folddoc f []       = nil
folddoc f [x]      = x
folddoc f (x:xs)   = f x (folddoc f xs)

spread             = folddoc (<+>)
stack              = folddoc (</>)
```

Preparing the XML-Application 2(3)

Further supportive functions:

```
-- An often recurring output pattern
bracket l x r   = group (text l <>
                         nest 2 (line <> x) <>
                         line <> text r)

-- Abbreviation of the alternative tree
-- layout function
showBracket' ts = bracket "[" (showTrees' ts) "]"

-- Filling up lines (using words out of the
-- Haskell Standard Lib.)
x <+/> y        = x <> (text " " :<|> line) <> y
fillwords       = folddoc (<+/>) . map text . words
```

Preparing the XML-Application 3(3)

fill, a variant of fillwords

⇝ collapses a list of documents to a single document.

```
fill []        = nil
fill [x]       = x
fill (x:y:zs)
  = (flatten x <+> fill (flatten y : zs)) :<|>
                (x </> fill (y : zs))
```

Application

Printing XML-documents (simplified syntax):

```
data XML              = Elt String [Att] [XML]
                        | Txt String

data Att              = Att String String

showXML x             = folddoc (<>) (showXMLs x)

showXMLs (Elt n a [])
     = [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c)
     = [text "<" <> showTag n a <> text ">" <>
        showFill showXMLs c <>
        text "</" <> text n <> text ">"]
showXMLs (Txt s)      = map text (words s)

showAtts (Att n v)
     = [text n <> text "=" <> text (quoted v)]
```

Application (Cont'd)

Continuation:

```
quoted s       = "\"" ++ s ++ "\""

showTag n a    = text n <> showFill showAtts a

showFill f [] = nil
showFill f xs
     = bracket "" (fill (concat (map f xs))) ""
```

1st XML Example

...for a given maximum line length of 30 letters:

```
<p
  color="red" font="Times"
  size="10"
>
  Here is some
  <em> emphasized </em> text.
  Here is a
  <a
    href="http://www.eg.com/"
  > link </a>
  elsewhere.
</p>
```

2nd XML Example

...for a given maximum line length of 60 letters:

```
<p color="red" font="Times" size="10" >
  Here is some <em> emphasized </em> text. Here is a
  <a href="http://www.eg.com/" > link </a> elsewhere.
</p>
```

3rd XML Example

...after dropping of `flatten` in `fill`:

```
<p color="red" font="Times" size="10" >
  Here is some <em>
    emphasized
  </em> text. Here is a <a
    href="http://www.eg.com/"
  > link </a> elsewhere.
</p>
```

...start and close tags are crammed together with other text
⇝ less beautifully than before.

Summing up: Why "prettier" than "pretty"?

The below pretty printer library proposed by John Hughes is widely recognized as a standard:

- John Hughes. The design of a pretty-printer library. In Johan Jeuring, Erik Meijer (Eds.), Advanced Functional Programming, Springer-V., LNCS 925, 53-96, 1995.

From a technical perspective, the library of John Hughes enjoys the following characteristics:

- There are two ways (horizontal and vertical) to concatenate documents, one of which
  - without unit (vertical)
  - with right-unit but no left-unit (horizontal)

Summing up (Cont'd)

Philip Wadler considers his "Prettier Printer" an improvement of John Hughes' pretty printer library.

From a technical perspective, a distinguishing feature of the "Prettier Printer" proposed by Philip Wadler is:

- There is only a single way to concatenate documents that is
  - associative
  - with a left-unit and a right-unit.

Moreover, John Hughes' pretty printer library

- consists of ca. 40% more code,
- is ca. 40% slower

as the "prettier printer" of Philip Wadler's proposal.

Summary of the Code 1(12)

Source: Philip Wadler. A Prettier Printer. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

```
infixr 5:<|>
infixr 6:<>
infixr 6 <>

data DOC = NIL
         | DOC :<> DOC
         | NEST Int DOC
         | TEXT String
         | LINE
         | DOC :<|> DOC

data Doc = Nil
         | String 'Text' Doc
         | Int 'Line' Doc
```

Summary of the Code 2(12)

```
nil                = NIL
x <> y             = x :<> y
nest i x           = NEST i x
text s             = TEXT s
line               = LINE

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
```

Summary of the Code 3(12)

```
layout Nil              = ""
layout (s 'Text' x) = s ++ layout x
layout (i 'Line' x) = '\n': copy i ' ' ++ layout x

copy i x                = [x | _ <- [1..i]]
```

Summary of the Code 4(12)

```
best w k x                  = be w k [(0,x)]

be w k []                   = Nil
be w k ((i,NIL):z)          = be w k z
be w k ((i,x :<> y) : z)
   = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)
   = s 'Text' be w (k+length s) z
be w k ((i,LINE) : z)      = i 'Line' be w i z
be w k ((i.x :<|> y) : z)
   = better w k (be w k ((i.x) : z))
                            (be w k (i,y) : z))

better w k x y
   = if fits (w-k) x then x else y
```

Summary of the Code 5(12)

```
fits w x | w<0         = False
fits w Nil             = True
fits w (s 'Text' x) = fits (w - length s) x
fits w (i 'Line' x) = True

pretty w x             = layout (best w 0 x)

-- Utility functions
x <+> y                = x <> text " " <> y
x </> y                = x <> line <> y

folddoc f []           = nil
folddoc f [x]          = x
folddoc f (x:xs)       = f x (folddoc f xs)
```

Summary of the Code 6(12)

```
spread         = folddoc (<+>)
stack          = folddoc (</>)

bracket l x r = group (text l <>
                      nest 2 (line <> x) <>
                      line <> text r)
x <+/> y       = x <> (text " " :<|> line) <> y

fillwords      = folddoc (<+/>) . map text . words

fill []        = nil
fill [x]       = x
fill (x:y:zs)
  = (flatten x <+> fill (flatten y : zs))
                 :<|> (x </> fill (y : zs))
```

Summary of the Code 7(12)

```
-- Tree example
data Tree             = Node String [Tree]

showTree (Node s ts) = group (text s <>
         nest (length s) (showBracket ts))

showBracket []        = nil
showBracket ts        = text "[" <>
         nest 1 (showTrees ts) <> text "]"

showTrees [t]         = showTree t
showTrees (t:ts)      = showTree t <> text "," <>
                           line <> showTrees ts
```

Summary of the Code 8(12)

```
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' []        = nil
showBracket' ts
  = bracket "[" (showTrees' ts) "]"

showTrees' [t]         = showTree t
showTrees' (t:ts)
  = showTree t <> text "," <> line <> showTrees ts
```

Summary of the Code 9(12)

```
tree        = Node "aaa"[ Node "bbbb"[ Node "ccc"[],
                                       Node "dd"[]
                        ],
                        Node "eee"[],
                        Node "ffff"[ Node "gg"[],
                                     Node "hhh"[],
                                     Node "ii"[]
                        ]
            ]

testtree w  = putStr(pretty w (showTree tree))
testtree' w = putStr(pretty w (showTree' tree))
```

Summary of the Code 10(12)

```
-- XML Example

data XML                     = Elt String [Att] [XML]
                               | Txt String

data Att                     = Att String String

showXML x                    = folddoc (<>) (showXMLs x)

showXMLs (Elt n a [])
  = [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c)
  = [text "<" <> showTag n a <> text ">" <>
                  showFill showXMLs c <>
                  text "</" <> text n <> text ">"]
showXMLs (Txt s)             = map text (words s)
```

Summary of the Code 11(12)

```
showAtts (Att n v)
  = [text n <> text "=" <> text (quoted v)]

quoted s       = "\"" ++ s ++ "\""

showTag n a    = text n <> showFill showAtts a

showFill f [] = nil
showFill f xs
  = bracket "" (fill (concat (map f xs))) ""
```

Summary of the Code 12(12)

```
xml =
 Elt "p"[Att "color" "red",
         Att "font" "Times",
         Att "size" "10"
        ] [ Txt "Here is some",
            Elt "em" [] [ Txt "emphasized"],
            Txt "text.",
            Txt "Here is a",
            Elt "a" [ Att "href" "http://www.eg.com/"]
                    [ Txt "link" ],
            Txt "elsewhere."
        ]

testXML w = putStr (pretty w (showXML xml))
```

Background Reading

On an early imperative "Pretty Printer:"

- Derek Oppen. Pretty-printing. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.

...and a functional realization of it:

- 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.

Background Reading (Cont'd)

Overview on the evolution of a Pretty Printer Library and origin of the development of the Prettier Printers proposed by Philip Wadler:

- 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 is implemented in the Glasgow Haskell Compiler:

- Simon Peyton Jones. Haskell pretty-printer library. 1997. `www.haskell.org/libraries/#prettyprinting`

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

Motivation

Hybrid systems are systems that are composed of

- continuous and
- discrete

components.

Mobile Robots

Mobile robots are special hybrid systems:

- From a physical perspective:
  - Continuous components: Voltage-controlled motors, batteries, range finders,...
  - Discrete components: Microprocessors, bumper switches, digital communication,...
- From a logical perspective:
  - Continuous notions: Wheel speed, orientation, distance from a wall,...
  - Discrete notions: Running into another object, receiving a message, achieving a goal,...

Objective of this Chapter

Desiging and implementing two

- imperative-style languages for controlling robots which will be done in terms of a simulation (in order to allow running the simulations at home without having to buy (possibly expensive) robots first).

This will deliver two examples of a

- domain specific language (DSL).

Simultaneously, it yields a nice application of the

- higher-order type (constructor) classes
  - Functor
  - Monad
  - Arrows

Reading for this Chapter

For Chapter 16.1:

- Paul Hudak. The Haskell School of Expression – Learning Functional Programming through Multimedia. Cambridge University Press, 2000. (Chapter 19, An Imperative Robot Language)

For Chapter 16.2:

- 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.

Remark:

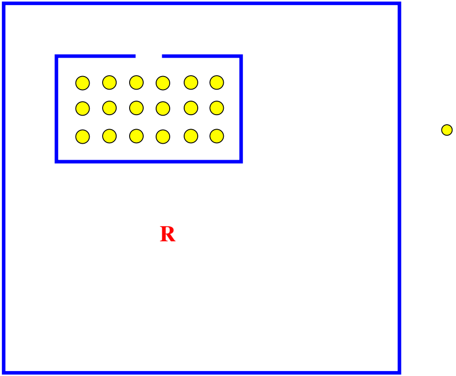
- Chapter 16.1 and 16.2 are independent of each other; they do not build on each other.

Chapter 16.1

An Imperative Robot Language

The World of Robots – Illustration (1)

Our robots' world:



...is two-dimensional with gold coins as treasures!.

The World of Robots – Illustration (2)

In more detail:

The world the robots live in

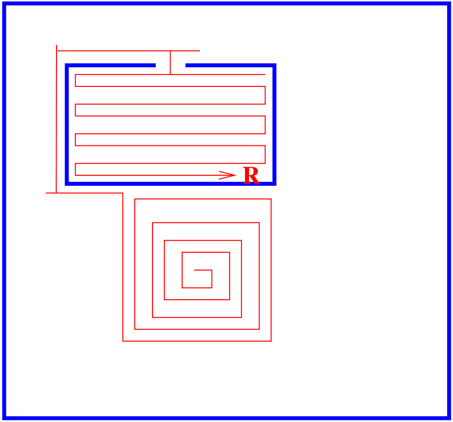
- is a finite two-dimensional grid of square form
  - equipped with walls
  - that might form rooms and might have doors
  - with placed gold coins on some grid points

The preceding illustration shows an example of a

- robot's world with one room full of gold coins: Eldorado!
- and a robot sitting in the centre of the world ready for exploring it!

The World of Robots – Illustration (3)

A robot's job:



...exploring the world, collecting treasures, leaving footprints!

The World of Robots – Illustration (4)

In more detail:

A robot's job

- is to explore its world, to collect the treasures in it, and to leave footprints of its exploration, i.e.,
  - to systematically stroll through its world, e.g., in the form of an outward-oriented spiral
  - picking up the gold coins it finds and saving them in its pocket
  - dropping gold coins at some grid points
  - marking its way with a colored pen

Objective

...enabling the robots to explore and shape their world!

In other words, we would like to write programs such as:

```
(1) drawSquare =
      do penDown
         move
         turnRight
         move
         turnRight
         move
         turnRight
         move
```

```
(2) moveToWall =
      while (isnt blocked)
      do move
```

```
(3) getRich =
      while (isnt blocked) $
        do move
           checkAndPickCoin
```

Modeling the World

Modeling the world our robots live and act in:

```
type Grid = Array Position [Direction]

type Position = (Int,Int)

data Direction = North | East | South | West
  deriving (Eq, Show, Enum)
```

Modeling the Robots (1)

The internal states of the robots are made up by:

1. Robot position
2. Robot orientation
3. Pen status (up or down)
4. Pen color
5. Placement of gold coins on the grid
6. Number of coins in the robot's pocket

Modeling the Robots (2)

Modeling the internal states of the robots:

```
data RobotState
  = RobotState
      {position :: Position
     , facing :: Direction
     , pen :: Bool
     , color :: Color
     , treasure :: [Position]
     , pocket :: Int
      }
    deriving Show
```

where

```
data Color = Black | Blue | Green | Cyan
               | Red | Magenta | Yellow | White
  deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

Remarks (1)

Note that the above definition takes advantage of Haskell's field-label syntax:

- Field labels (here `position`, `facing`, `pen`, `color`, `treasure`, `pocket`) allow access to components by names instead of position without necessitating specific selector functions.

Remarks (2)

Robot states could have been equivalently be defined without referring to field label syntax:

```
data RobotState
  = RobotState
      Position
      Direction
      Bool
      Color
      [Position]
      Int
  deriving Show
```

...losing the advantage of accessing fields by names.

Remarks (3)

Illustrating the usage of field labels: Generating, accessing, modifying values of state components.

Example 1: Generating field values

The definition

```
s1 = RobotState (0,0) East True Green
             [(2,3),(7,9),(12,42)] 2 :: RobotState
```

is equivalent to

```
s2 = RobotState { position = (0,0)
                , facing   = East
                , pen      = True
                , color    = Green
                , treasure = [(2,3),(7,9),(12,42)]
                , pocket   = 2
                } :: RobotState
```

Remarks (4)

Advantages of using field label syntax:

- It is more "informative."
- The order of fields gets irrelevant.

  For example: The definition of s3

```
s3 = RobotState
        { position = (0,0)
        , pocket   = 2
        , pen      = True
        , color    = Green
        , treasure = [(2,3),(7,9),(12,42)]
        , facing   = East
        } :: RobotState
```

  is equivalent to that of s2.

Remarks (5)

Example 2: Accessing field values

```
position s2  ->>  (0,0)
treasure s3  ->>  [(2,3),(7,9),(12,42)]
color    s3  ->>  Green
```

Example 3: Modifying field values

```
s3 { position = (22,43), pen = False }
 ->> RobotState { position = (22,43)
                , facing = East
                , pen = False
                , color = Green
                , treasure = [(2,3),(7,9),(12,42)]
                , pocket = 2
                } :: RobotState
```

Remarks (6)

Example 4: Using field names in patterns

```
jump (RobotState { position = (x,y) }) = (x+2,y+1)
```

Robots as a Member of Type Class Monad

Defining Robot as an algebraic data type

```
newtype Robot a
 = Robot (RobotState -> Grid
            -> Window -> IO (RobotState,a))
```

...allows making Robot an instance of type class Monad:

```
instance Monad Robot where
 Robot sf0 >>= f = Robot $ \s0 g w -> do
                        (s1,a1) <- sf0 s0 g w
                         let Robot sf1 = f a1
                         (s2,a2) <- sf1 s1 g w
                         return (s2,a2)
 return a        = Robot (\s _ _ -> return(s,a))
```

Remarks (1)

Note that

```
instance Monad Robot where
 return a         = Rob (\s _ _ -> return(s,a))
 Rob sf0 >>= f    = Rob $ \s0 g w -> do
                         (s1,a1) <- sf0 s0 g w
                         let Rob sf1 = f a1
                         (s2,a2) <- sf1 s1 g w
                         return (s2,a2)
```

requires function application "$", not function composition "." (For clarity, Robot has been replaced by Rob (cp. next slide)).

Remarks (2)

The Window argument

```
newtype Robot a
  = Rob (RobotState -> Grid
                -> Window -> IO (RobotState,a))
```

...allows to specify the window, in which the graphics is displayed.

Robots – Simulation and Control

The implementation environment:

```
module Robot where
import Array
import List
import Monad
import SOEGraphics
import Win32Misc (timeGetTime)
import qualified GraphicsWindows as GW (getEvent)
```

Note:

- Graphics, SOEGraphics are two commonly used graphics libraries being Windows-compatible.
- Double-check the SOE homepage at haskell.org/soe regarding the availability of the modules SOEGraphics and GraphicsWindows.

IRL – The Imperative Robot Language (1)

Key insight:

- Taking state as input
- Possibly querying the state in some way
- Returning a possibly modified state

...makes the imperative nature of IRL commands obvious.

IRL – The Imperative Robot Language (2)

IRL commands and their implementation:

- Commands not related to graphics:

```
right, left :: Direction -> Direction

right d = toEnum (succ (mod (fromEnum d) 4))
left d  = toEnum (pred (mod (fromEnum d) 4))
```

- Supporting functions for updating and querying states:

```
updateState :: (RobotState -> RobotState)
                                  -> Robot ()
updateState u = Robot (\s _ _ -> return (u s, ()))

queryState :: (RobotState -> a) -> Robot a
queryState q = Robot (\s _ _ -> return (s, q s))
```

The Type Class Enum (1)

...of the Standard Prelude:

```
class Enum a where
 succ, pred     :: a -> a
 toEnum         :: Int -> a
 fromEnum       :: a -> Int
 enumFrom       :: a -> [a]                -- [n..]
 enumFromThen   :: a -> a -> [a]           -- [n,n'..]
 enumFromTo     :: a -> a -> [a]           -- [n..m]
 enumFromThenTo :: a -> a -> a -> [a]  -- [n,n'..m]

 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
```

The Type Class Enum (2)

The following equivalences hold:

```
enumFrom n              ~  [n..]
enumFromThen n n'       ~  [n,n'..]
enumFromTo n m          ~  [n..m]
enumFromThenTo n n' m   ~  [n,n'..m]
```

Example:

```
data Color = Red | Orange | Yellow | Green
             | Blue | Indigo | Violet

instance Enum Color where
  ...

[Red..Green]      ->>  [Red, Orange, Yellow, Green]
[Red, Yellow..]   ->>  [Red, Yellow, Blue, Violet]
fromEnum Blue     ->>  4
toEnum 3          ->>  Green
```

IRL – The Imperative Robot Language (3)

- Commands for robot orientation:

```
turnLeft :: Robot ()
turnLeft =
 updateState (\s -> s {facing = left (facing s)})

turnRight :: Robot ()
turnRight =
 updateState (\s -> s {facing = right (facing s)})

turnTo :: Direction -> Robot ()
turnTo d = updateState (\s -> s {facing = d})

direction :: Robot Direction
direction =  queryState facing
```

IRL – The Imperative Robot Language (4)

- Commands for blockade checking:

```
blocked :: Robot Bool
blocked =
 Robot $ \s g _ ->
  return(s, facing s 'notElem' (g 'at' position s))
```

IRL – The Imperative Robot Language (5)

- Commands for moving a robot:

```
move :: Robot ()
move =
 cond1 (isnt blocked)
  (Rob $ \s _ w -> do
    let newPos = movePos (position s) (facing s)
    graphicsMove w s newPos
    return (s {position = newPos}, ())
   )
movePos :: Position -> Direction -> Position
movePos (x,y) d
        = case d of
            North -> (x,y+1)
            South -> (x,y-1)
            East  -> (x+1,y)
            West  -> (x-1,y)
```

IRL – The Imperative Robot Language (6)

- Commands for using the pen:

```
penUp :: Robot ()
penUp = updateState (\s -> s {pen = False})

penDown :: Robot ()
penDown = updateState (\s -> s {pen = True})

setPenColor :: Color -> Robot ()
setPenColor c = updateState (\s -> s {color = c})
```

IRL – The Imperative Robot Language (7)

- Commands for handling coins:

```
onCoin :: Robot Bool
onCoin =  queryState (\s ->
               position s 'elem' treasure s)

coins :: Robot Int
coins =  queryState pocket
```

IRL – The Imperative Robot Language (8)

- More commands for handling coins:

```
pickCoin :: Robot ()
pickCoin =
 cond1 onCoin
  (Robot $ \s _ w ->
     do eraseCoin w (position s)
        return (s {treasure =
                     position s 'delete' treasure s,
                    pocket = pocket s+1}, ())
  )
```

IRL – The Imperative Robot Language (9)

- More commands for handling coins:

```
dropCoin :: Robot ()
dropCoin =
 cond1 (coins >* return 0)
  (Robot $ \s _ w ->
     do drawCoin w (position s)
        return (s {treasure =
                             position s : treasure s,
                          pocket = pocket s-1}, ())
  )
```

Logic and Control (1)

- Logic and control functions:

```
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 ())

while :: Robot Bool -> Robot () -> Robot ()
while p b = cond1 p (b >> while p b)

(||*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 ||* b2 = do p <- b1
               if p then return True
                    else b2
```

Logic and Control (2)

- Logic and control functions (cont'd):

```
(&&*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 &&* b2 = do p <- b1
               if p then b2
                    else return False

isnt :: Robot Bool -> Robot Bool
isnt =  liftM not

(>*) :: Robot Int -> Robot Int -> Robot Bool
(>*)       =  liftM2 (>)

(<*) :: Robot Int -> Robot Int -> Robot Bool
(<*)       =  liftM2 (<)
```

Logic and Control (3)

The higher-order functions `liftM` and `liftM2` are defined in the library `Monad` (as well as `liftM3`,...,`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')
```

Logic and Control (4)

It is worth noting:

- Basing the implementations of `isnt`, `(>*)` and `(<*)` on `liftM` and `liftM2` allows to dispense the usage of special lift funcions.
- No basing of the implementations of `(||*)` and `(&&*)` on `liftM2` in order to avoid (unnecessary) strictness in their second arguments.

Further Data Structures

```
colors :: Array Int Color
colors = array (0,7)
           [(0,Black),(1,Blue),(2,Green),(3,Cyan),
            (4,Red),(5,Magenta),(6,Yellow),(7,White)]
```

where (as a reminder!)

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
  deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

Note:

- `Color` is defined as in the library `Graphics`.
- Equivalently we could have defined more concisely:

  ```
  colors :: Array Int Color
  colors = array (0,7) (zip [0..7] [Black..White])
  ```

Shaping the Robots' Initial World g0

The robots' world is a grid of type `Array`:

```
type Grid = Array Position [Direction]
```

We can access the grid points using:

```
at :: Grid -> Position -> [Direction]
at =  (!)
```

The size of the inital grid g0 is given by:

```
size :: Int
size =  20
```

with

- centre `(0,0)` and
- corners `(size,size)`, `((-size),size)`, `((-size),(-size))` and `(size,(-size))`.

The Initial World g0 (1)

...and the 4 surrounding walls (no walls inside):

- Inner points of g0 are given by:

```
interior = [North, South, East, West]
```

- Extremal points on the grid borders (north border, north-east corner, etc.) are given by:

```
nb  = [South, East, West]  -- nb: north border
sb  = [North, East, West]
eb  = [North, South, West]
wb  = [North, South, East] -- wb: west border
nwc = [South, East]  -- nwc: northwest corner
nec = [South, West]
swc = [North, East]
sec = [North, West]  -- sec: southeast corner
```

The Initial World g0 (2)

This allows:

...enumerating inner and border grid points using a list comprehension:

```
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]
```

The new World g1 that extends World g0 (1)

...evolves from building new walls using the array library functions (//):

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

Example: Application of (//)

- Reversing the positions of "black" und "white" in `colors`:

```
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 Integer Color
```

The new World g1 that extends World g0 (2)

Supporting functions for building new walls:

```
-- Building horizontal and vertical walls
mkHorWall, mkVerWall ::
 Int -> Int -> Int -> [(Position,[Direction])]

-- Building west/east-oriented walls
-- leading from (x1,y) to (x2,y)
mkHorWall x1 x2 y
 = [((x,y), nb) | x <- [x1..x2]] ++
   [((x,y+1), sb) | x <- [x1..x2]]

-- Building north/south-oriented walls
-- leading from (x,y1) to (x,y2)
mkVerWall y1 y2 x
 = [((x,y), eb) | y <- [y1..y2]] ++
   [((x+1,y), wb) | y <- [y1..y2]]
```

The new World g1 that extends World g0 (3)

World g1 evolves from world g0 by

- building a west/east-oriented wall leading from (-5,10) to (5,10):

```
g1 :: Grid
g1 =  g0//mkHorWall (-5) 5 10
```

The World g2 that extends g0 (1)

Supporting functions for building a "room:"

```
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
```

Note:

- The above function creates two field entries for each of the four inner corners.
- After creation the value of these entries are still undefined.
- Using the function accum allows initializing these entries on-the-fly of their creation:

```
accum :: (Ix a) => (b -> c -> b)
                 -> Array a b -> [(a,c)] -> Array a b
```

The World g2 that extends g0 (2)

Recall the function accum:

```
accum :: (Ix a) => (b -> c -> b)
  -> Array a b -> [(a,c)] -> Array a b
```

The function accum

- is quite similar to the function (//).
- in case of replicated entries the function of the first argument is used for resolving conflicts.
- the List-library function intersect is suitable for this for the case of our example:

  Example:

  ```
  [South, East, West] 'intersect'
      [North, South, West] ->> [South, West]
  ```

  which corresponds to a northeast corner.

The World g2 that extends g0 (3)

Example: Building a room with (-10,5) as lower left corner and (-5,10) as upper right corner

- using `accum` und `intersect`.

World g2 then extends world g0:

```
g2 :: Grid
g2 =  accum intersect g0 (mkBox (-15,8) (2,17))
```

The World g3 that extends g2

Continuing the example: Adding a door (to the middle of the top wall of the room)

- using accum und union.

World g2 evolves to world g3:

```
g3 :: Grid
g3 =  accum union g2 [((-7,17), interior),
                      ((-7,18), interior)]
```

Animation: Robot Graphics (1)

Animation

- by means of incrementally updating the world.

To this end we make use of the function:

```
drawLine :: Window -> Color
                   -> Point -> Point -> IO ()
drawLine w c p1 p2
  = drawInWindowNow w (withColor c (line p1 p2))
```

which makes use of the `Graphics`-library function `drawInWindowNow`.

Animation: Robot Graphics (2)

The incremental update of the world must ensure

- absence of interferences of graphics actions.

To this end we assume:

1. Grid points are 10 pixels apart.
2. Wall are drawn halfway between grid points.
3. Lines drawn by a robot's pen directly connects two grid points.
4. Coins are drawn as yellow circles just to the above and to to the left of a grid point.
5. Erasing coins is done by drawing black circles over already existing yellow ones.

Animation: Robot Graphics (3)

Using the below top level constants ensures the absence of interferences:

```
d            :: Int
d            =  5         -- half the distance
                          -- between grid points

wc, cc       :: Color
wc           =  Blue      -- color of walls
cc           =  Yellow    -- color of coins

xWin, yWin :: Int
xWin         =  600
yWin         =  500
```

Animation in Action (1)

Putting it all together.

User-control of program progress by the program's awaiting the user's hitting the spacebar:

```
spaceWait   :: Window -> IO ()
spaceWait w = do k <- getKey w
                 if k==' ' then return ()
                           else spaceWait w
```

Animation in Action (2)

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
```

Animation in Action (3)

Intuitively, `runRobot` causes:

- Opening a window
- Drawing a grid
- Drawing the coins
- Waiting for the user to hit the spacebar
- Continuing running the program with starting state `s` and grid `g`

Animation in Action (4)

Fixing a suitable starting state:

```
s0 :: RobotState
s0 =  RobotState {position = (0,0)
                 , pen = False
                 , color = Red
                 , facing = North
                 , treasure = tr
                 , pocket = 0
                  }

tr :: [Position]
tr =  [(x,y) | x <- [-13,-11..1], y <- [9,11..15]]
```

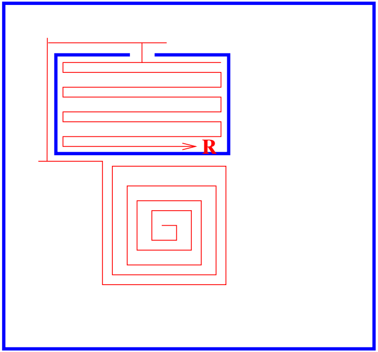
...i.e., all coins are placed inside of the room of grid g3.

Animation in Action (5)

Last but not least:

```
main = runRobot spiral s0 g0
```

...leads to the "spiral" example discussed at the beginning of this chapter:



Additional Supporting Functions (1)

For drawing a grid:

```
drawGrid :: Window -> Grid -> IO ()
drawGrid w wld =
 let (low@(xMin,yMin),hi@(xMax,yMax)) = bounds wld
     (x1,y1)                          = trans low
     (x2,y2)                          = trans hi
 in
  do
   drawLine w wc (x1-d,y1+d) (x1-d,y2-d)
   drawLine w wc (x1-d,y1+d) (x1+d,y2+d)
   sequence_ [drawPos w (trans (x,y)) (wld 'at' (x,y))
          | x <- [xMin..xMax], y <- [yMin..yMax]]
```

Additional Supporting Functions (2)

```
drawPos :: Window -> Point -> [Direction] -> IO ()
drawPos x (x,y) ds
  = do if North 'notElem' ds
          then drawLine w wc (x-d,y-d) (x+d,y-d)
          else return ()
       if East 'notElem' ds
          then  drawLine w wc (x+d,y-d) (x+d,y+d)
          else return ()
```

Additional Supporting Functions (3)

For dropping and erasing coins:

```
drawCoins        :: Window -> RobotState -> IO ()
drawCoins w s  = mapM_ (drawCoin w) (treasure s)

drawCoin         :: Window -> Position -> IO ()
drawCoin w p   =
 let (x,y) = trans p
 in drawInWindowNow w
    (withColor cc (ellipse (x-5,y-1) (x-1,y-5)))

eraseCoin        :: Window -> Position -> IO ()
eraseCoin w p  =
 let (x,y) = trans p
 in drawInWindowNow w
     (withColor Black (ellipse (x-5,y-1) (x-1,y-5)))
```

Further Supporting Functions (4)

```
graphicsMove :: Window -> RobotState
                              -> Position -> IO ()
graphicsMove w s newPos
 = do
    if pen s
       then
        drawLine w (color s) (trans (position s))
                              (trans newPos)
       else return ()
    getWindowTick w
```

Further Supporting Functions (5)

```
trans         :: Position -> Point
trans (x,y) = (div xWin 2+2*d*x, div yWin 2-2*d*y)

getWindowTick :: Window -> IO ()
-- causes a short delay after each robot move

bounds :: Ix a => Array a b -> (a,a)
-- from the Array-library; yields the bounds
-- of an array argument
```

Chapter 16.2

Robots on Wheels

Outline

A second simulation of

- mobile robots by means of functional reactive programming.

This time we will make use of

- the type class `Arrows` that is another example of a higher-order type class that generalizes the concept of a monad.

The Scenario (1)

Mobile robots are assumed to be configured as follows:

"Roboter haben zwei Räder, die von je einem Motor unabhängig voneinander angetrieben werden. Die relative Geschwindigkeit der Räder zueinander legt die Richtung fest, in die sich der Roboter bewegt. Bewegen sich beide Räder gleich schnell, fährt der Roboter geradeaus.

Roboter verfügen über unterschiedliche Sensoren wie etwa einen Berührungssensor, ob der Roboter auf ein Hindernis trifft, Entfernungssensor, um die Entfernung zum nächstgelegenen Objekt in einer bestimmten Richtung zu bestimmen (in der Folge nur für vorwärts, rückwärts, rechts und links), einen Objektsensor, der die aktuelle Position aller anderen Roboter liefert und ggf. die Position weiterer beweglicher Objekte wie z.B. von Bällen innerhalb einer bestimmten Entfernung vom Roboter (Objektsensoren etwa reali-

The Scenario (2)

siert durch visuelle Subsysteme oder Kommunikationssysteme zum Austausch von Koordinaten).

Roboter haben eine eindeutige Kennung (ID) und ggf. weitere (bei Bedarf) eingeführte Sensoren."

Motivation of the Scenario: Robot Soccer

Robot soccer provides a motivation for such a scenario:

"Schreibe ein Programm "robocup soccer" wie folgt:

Erzeuge mithilfe von Wandsegmenten zwei Tore an jedem Spielfeldende.

Bestimme die Anzahl von Spielern pro Mannschaft und schreibe Steuerungsprogramme für diese Spieler. Zweckmäßigerweise sind die Steuerungsprogramme generisch für unterschiedliche Spielertypen, etwa für Torhüter, Stürmer, Verteidiger.

Erzeuge eine initiale Welt, in der der Ball am Anstoßpunkt im Mittelkreis liegt und jeder der Spieler strategisch günstig in der jeweils eigenen Spielhälfte und außerhalb des Mittelkreises positioniert ist. Jede Mannschaft mag die gleichen oder auch unterschiedliche Steuerungsprogramme nutzen."

Simulation Code for "Robots on Wheels"

...can be down-loaded at the Yampa homepage at

`www.haskell.org/yampa`

In the following we will consider some code snippets.

Structure of the Program

```
module MyRobotShow where
 import AFrob
 import AFrobRobotSim

 main :: IO ()
 main = runSim (Just world) rcA rcB

 world :: WorldTemplate
 world = ...

 rcA :: SimbotController -- controller for simbot A's
 rcA = ...

 rcB :: SimbotController -- controller for simbot B's
 rcB = ...
```

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
```

Robot Control

```
type SimbotController =
      SimbotProperties -> SF SimbotInput SimbotOutput

Class HasRobotProperties i where
 rpType      :: i -> RobotType    -- Type of robot
 rpId        :: i -> RobotId      -- Identity of robot
 rpDiameter :: i -> Length        -- Distance between wheels
 rpAccMax    :: i -> Acceleration -- Max translational acc
 rpWSMax     :: i -> Speed        -- Max wheel speed

type RobotType = String
type RobotId   = Int
```

Robot Simulation in Action

Running the robot simulation:

```
runSim :: Maybe WorldTemplate
               -> SimbotController
                    -> SimbotController -> IO ()
```

Robot Control

```
rcA :: SimbotController
rcA rProps =
  case rrpId rProps of
    1 -> rcA1 rProps
    2 -> rcA2 rProps
    3 -> rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
```

Robot Actions: Control Programs (1)

A stationary robot:

```
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)
```

Classes of Robots (1)

- Usually, there are different types of robots with different features (2 wheels, 3 wheels, camera, sonar, speaker, blinker, etc.)
- The kind of a robot is fixed by its input and output types.

The kind of robots is encoded in input and output classes together with the functions operating on them.

Kinds of Robots (2)

Input classes and functions operating on them:

```
class HasRobotStatus i where
  rsBattStat :: i -> BatteryStatus -- Current battery
                                   -- status
  rsIsStuck  :: i -> Bool          -- Currently stuck
                                   -- or not stuck

data BatteryStatus = BSHigh | BSLow | BSCritical
     deriving (Eq, Show)

-- derived event sources:
rsBattStatChanged  :: HasRobotStatus i
                        => SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i => SF i (Event ())
rsBattStatCritical :: HasRobotStatus i => SF i (Event ())
rsStuck            :: HasRobotStatus i => SF i (Event ())
```

Classes of Robots (3)

```
class HasOdometry where
  odometryPosition :: i -> Position2 -- Current
                                     -- position
  odometryHeading  :: i -> Heading   -- Current
                                     -- heading

class HasRangeFinder i where
  rfRange    :: i -> Angle -> Distance
  rfMaxRange :: i -> Distance

-- derived range finders:
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance
```

Classes of Robots (4)

```
class HasAnimateObjectTracker i where
 aotOtherRobots :: i -> [(RobotType, Angle, Distance)]
 aotBalls       :: i -> [(Angle, Distance)]

class HasTextualConsoleInput i where
 tciKey :: i -> Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i =>
                   Maybe Char -> SF i (Event Char)
tciKeyDown    :: HasTextualConsoleInput i =>
                   SF i (Event Char)
```

Classes of Robots (5)

Output classes and functions operating on them:

```
class MergeableRecord o => HasDiffDrive o where
 ddBrake :: MR o -- Brake both wheels
 ddVelDiff :: Velocity -> Velocity
                              -> MR o -- Set wheel
                                      -- velocities
 ddVelTR   :: Velocity -> RotVel
                              -> MR o -- Set veloc.
                                      -- and rotat.

class MergeableRecord o =>
  HasTextConsoleOutput o where
    tcoPrintMessage :: Event String -> MR o
```

Arrows and Mobile Robots

SF is an instance of class Arrow:

```
SF a b      = Signal a -> Signal b

Signal a  = Time -> a

type Time = Double
```

Note:

- Values of type SF are signal transformers resp. signal functions; therefore the name SF.

Chapter 16.3

More on the Background of FRP

Origins of FRP

The origins of functional reactive programming (FRP) lie in functional reactive animation (FRAn):

- 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 PLILP/ALP'98, Springer-Verlag, 1998.

Seminal Works on FRP

Seminal works on function 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.
  http://www.haskell.org/frp/manual.html
- John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. Yale FRP User's Manual. Department of Computer Science, Yale University, January 2001.
- Henrik Nilsson, Antony Courtney, John Peterson. Functional Reactive Programming, Continued. In Proceedings of the ACM SIGPLAN'02 Haskell Workshop, October 2002.

Applications of FRP (1)

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 Reactive Robotics (FRob):

- Izzet Pembeci, Henrik Nilsson, Gregory Hager. Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'02), October 2002.
- John Peterson, Gregory Hager, Paul Hudak. A Language for Declarative Robotic Programming. In Proceedings of the International Conference on Robotics and Automation, 1999.

Applications of FRP (2)

On Functional Vision Systems (FVision):

- Alastair Reid, John Peterson, Gregory Hager, Paul Hudak. Prototyping Real-Time Vision Systems: An Experiment in DSL Design. In Proceedings of the International Conference on Software Engineering, May 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'01 International Conference on Functional Programming (ICFP 2001), ACM Press, 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), ACM Press, January 2002.

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)

- Alastair Reid, John Peterson, Gregory 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.
- 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.

Chapter 16: Further Reading (7)

- 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 and Prospectives

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

- Kapitel 21, Massiv Parallele Programme

  Peter Pepper, Petra Hofstedt. Funktionale Programmierung, Springer-V., 2006. (In German).

Parallelism in Imperative Languages

Predominant:

- Data-parallel Languages (e.g. High Performance Fortran)
- Libraries (PVM, MPI) $\rightsquigarrow$ Message Passing Model (C, C++, Fortran)

Parallelism in Functional Languages

Predominant:

- Implicit (expression) parallelism
- Explicit parallelism
- Algorithmic skeletons

Implicit Parallelism

...also known as expression parallelism.

Let `f(e1,...,en)` be 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.
- Most important disadvantage: Results often unsatisfying; e.g. granularity, load distribution, etc. is not taken into account.

Summing up, expression parallelism is

- easy to detect (i.e., for the compiler) but hard to fully exploit.

Explicit Parallelism

By means of

- Introducing meta-statements (e.g. to control the data and load distribution, communication)
- Most important advantage: Often very good results thanks to explicit hands-on control of the programmer.
- Most important disadvantage: High programming effort and loss of functional elegance.

Algorithmic Skeletons

...a compromise between

- explicit imperative parallel programming
- implicit functional expression parallelism

In the following

We assume a scenario with

- Massively parallel systems
- Algorithmic skeletons

Massively Parallel Systems

...characterized by

- large number of processors with
  - local memory
  - communication by message exchange
- MIMD-Parallel Processor Architecture (Multiple Instruction/Multiple Data)

Here we restrict ourselves to:

- SPMD-Programming Style (Single Program/Multiple Data)

Algorithmic Skeletons

Algorithmic skeletons

- represent typical patterns for parallelization (Farm, Map, Reduce, Branch&Bound, Divide&Conquer,...)
- are easy to instantiate for the programmer
- allow parallel programming at a high level of abstraction

Implementation of Algorithmic Skeletons

...in functional languages

- by special higher-order functions
- with parallel implementation
- embedded in sequential languages

Advantages:

- Hiding of parallel implementation details in the skeleton
- Elegance and (parallel) efficiency for special application patterns.

Example: Parallel Map on Distributed List

Consider the higher-order function map on lists:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)
```

Observation:

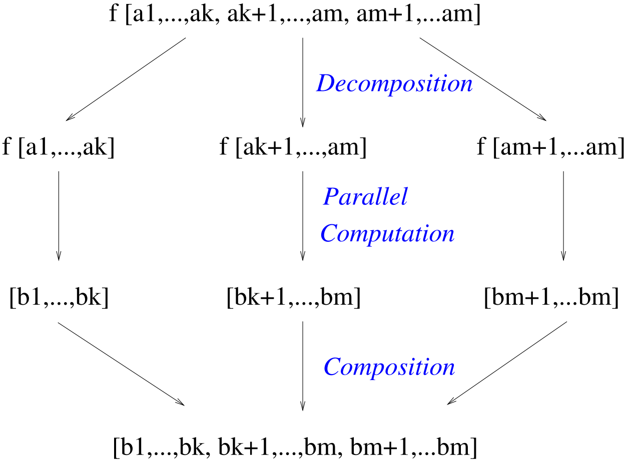
- Applying f to a list element does not depend on other list elements.

Obviously:

- Dividing the list into sublists followed by parallel application of map to the sublists: 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

- so-called data-parallel skeletons.

Note the difference:

- Data-parallelism: Supposes an a priori distribution of data on different processors.
- Task-parallelism: Processes and data to be distributed are not known a priori, hence dynamically generated.

Programming of a Parallel Application

...using algorithmic skeletons:

- Recognizing problem-inherent parallelism.
- Selecting an adequate data distribution (granularity).
- Selecting a suitable skeleton from a library.
- Instantiating a problem-specific skeleton.

Remark:

- Some languages (e.g. Eden) support the implementation of skeletons (in addition to those which might be provided by a library).

Data Distribution on Processors

...is crucial for

- the structure of the complete algorithm
- efficiency

The hardness of the distribution problems depends on

- Independence of all data elements (like in the map-example): Distribution is easy.
- Independence of subsets of data elements.
- Complex dependences of data elements: Adequate distribution is challenging.

Auxiliary means:

- So-called covers (investigated by various researchers).

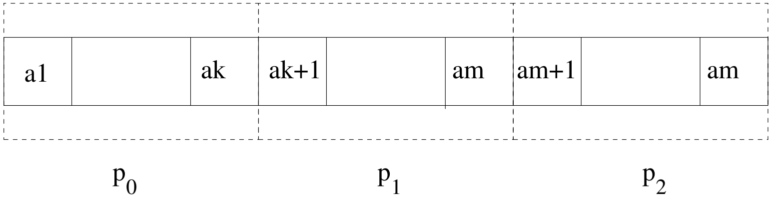
Covers

...describe the

- decomposition and communication pattern of a data structure.

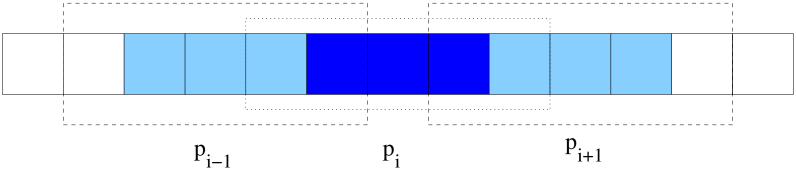
Illustration of a Simple List Cover

Distributing a list on 3 processors p_0, p_1, and p_2:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

Illustration of a List Cover with Overlapping Elements



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

General Cover Structure

```
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
```

It is required:

```
glue . split = id
```

Note: The above code snippet is not (valid) Haskell.

Implementation in a Programming Language

Implementing covers requires support for

- the specification of covers.
- the programming of algorithmic skeletons on covers.
- the provision of often used skeletons in libraries.

It is

- currently a hot research topic in functional programming.

Last but not least

Implementing skeletons

- by message passing via skeleton hierarchies.

Chapter 17.1: Further Reading (1)

- 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.
- 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)
- Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*. In Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 295-308, 1996.

Chapter 17.1: Further Reading (4)

- 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.
- Robert F. Pointon, Philip W. Trinder, Hans-Wolfgang Loidl. *The Design and Implementation of Glasgow Distributed Haskell*. In Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), LNCS 2011, Springer-V., 53-70, 2000.
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.3, Parallel Algorithms)

Chapter 17.1: Further Reading (5)

- 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

Haskell for “Real World Programming”

"Real World" Haskell (1)

Haskell these days provides considerable, mature, and stable support for:

- Systems Programming
- (Network) Client and Server Programming
- Data Base and Web Programming
- Multicore Programming
- Foreign Language Interfaces
- Graphical User Interfaces
- File I/O and filesystem programming
- Automated Testing, Error Handling, and Debugging
- Performance Analysis and Tuning
- ...

"Real World" Haskell (2)

This support, which comes mostly in terms of

- sophisticated libraries

makes Haskell a reasonable choice for addressing and solving

- Real World Problems

since such a choice depends much on the ability and support a programming language (environment) provides for linking and connecting to the "outer world."

Chapter 17.2: Further Reading (1)

- 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 (2)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 17, Interfacing with C: The FFI; Chapter 19, Error Handling; Chapter 20, Systems Programming in Haskell; Chapter 21, Using Data Bases; Chapter 22, Extended Example: Web Client Programming; Chapter 23, GUI Programming with gtk2hs; Chapter 24, Concurrent and Multicore Programming; Chapter 27, Sockets and Syslog; Chapter 25, Profiling and Optimization; Chapter 28, Software Transactional Memory)

- 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 (3)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 19, Agenten und Prozesse; Kapitel 20, Graphische Schnittstellen (GUIs))
- "Haskell community." *Hackage: A Repository for Open Source Haskell Libraries*. `hackage.haskell.org`
- "Haskell community." *Haskell wiki*. `haskell.org/haskellwiki/Applications_and_libraries`
- Useful search engines: Hoogle and Hayoo. `www.haskell.org/hoogle`, `holumbus.fh-wedel.de/hayoo/hayoo.html`

Chapter 18

Conclusions and Prospectives

Research Venues, Research Topics, and More

...for functional programming and functional programming languages:

- Research/publication/dissemination venues
  - Conference and Workshop Series
  - Archival Journals
  - Summer Schools
- Research Topics
- Functional Programming in the Real World

Relevant Conference and Workshop Series

For functional programming:

- Annual ACM SIGPLAN International Conference on Functional Programming (ICFP) Series, since 1996.
- Annual 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"

Contest Announcement at ICFP 2012 (1)

The ICFP Programming Contest 2012 is the 15th instance of the annual programming contest series sponsored by The ACM SIGPLAN International Conference on Functional Programming. This year, the contest starts at 12:00 July 13 Friday UTC and ends at 12:00 July 16 Monday UTC. There will be a lightning division, ending at 12:00 July 14 Saturday UTC.

The task description will be published at `icfpcontest2012.wordpress.com/task` when the contest starts. Solutions to the task must be submitted online before the contest ends. Details of the submission procedure will be announced along with the contest task.

This is an open contest. Anybody may participate except for the contest organisers and members of the same group as the contest chairs. No advance registration or entry fee is required.

Contest Announcement at ICFP 2012 (2)

Any programming language(s) may be used as long as the submitted program can be run by the judges on a standard Linux environment with no network connection. Details of the judges environment will be announced later.

There will be cash prizes for the first and second place teams, the team winning the lightning divison, and a discretionary judges prize. There may also be travel support for the winning teams to attend the conference. (The prizes and travel support are subject to the budget plan of ICFP 2012 pending approval by ACM.)...

More on ICFP 2012, Copenhagen, DK, 10-12 Sep 2012:
http://icfpconference.org/icfp2012/cfp.html

Contest Announcement at ICFP 2013

- This year's contest will take place from August 8, 2013 to August 11, 2013.
- Detailed information on it will be announced soon.
- Stay tuned for news on this year's contest at `research.microsoft.com/en-us/events/icfpcontest2013`

More on ICFP 2013, Boston, MA, 25-27 Sep 2013:
`http://icfpconference.org/icfp2013`

Functional Programming 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.
- Yaron Minsky. OCaml for the Masses. Communications of the ACM, 54(11):53-58, 2011.
- Haskell in Industry and Open Source: www.haskell.org/haskellwiki/Haskell_in_industry

Recall Edsger W. Dijkstra's Prediction

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.(*)

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

(*) Quote from: Introducing a course on calculi. Announcement of a lecture course at the University of Texas at Austin, 1995.

In the Words of John Carmack

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

John Carmack

Chapter 18: Further Reading (1)

- 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.
- Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. Journal of Functional Programming 7(2):163-182, 1997.

Chapter 18: Further Reading (2)

- 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. (Chapter 25, Profiling and Optimization)
- Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

Chapter 18: Further Reading (3)

- David A. Turner. *Total Functional Programming*. Journal of Universal Computer Science 10(7):751-768, 2004.
- Marcos Viera, S. Doaitse Swierstra, Wouter S. Swierstra. *Attribute Grammars fly First Class: How do we do Aspect Oriented Programming in Haskell*. In Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009), 245-256, 2009.
- "Haskell community." *Haskell in Industry and Open Source*. `www.haskell.org/haskellwiki/Haskell_in_industry`

Bibliography

Reading

...for deepened and independent studies.

- I Textbooks
- II Monographs
- III Volumes
- IV Articles
- V Haskell 98 – Language Definition
- V The History of Haskell

I Textbooks (1)

- 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.
- Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised edition, North Holland, 1984.

I Textbooks (2)

- Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- Richard E. Bellman, Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1957.
- Richard Bird. *Introduction to Functional Programming using Haskell*. 2nd edition, Prentice-Hall, 1998.
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
- William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011.

I Textbooks (3)

- 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.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.
- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004.

I Textbooks (4)

- Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999.
- Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
- Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions – An Introduction to Computer Science using Scheme*. Brooks/Cole Publishing Company, 1999.
- John O'Donnell, Cordelia Hall, Rex Page. *Discrete Mathematics Using a Computer*. Springer-V., 2nd edition, 2006.

I Textbooks (5)

- Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004.
- Peter Henderson. *Functional Programming: Application and Implementation*. Prentice Hall, 1980.
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
- Mark P. Jones, Alastair Reid et al. (Eds.). *The Hugs98 User Manual*. www.haskell.org/hugs

I Textbooks (6)

- 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.
- 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.
- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2nd edition, 2011.

I Textbooks (7)

- 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.
- Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008.

I Textbooks (8)

- 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.
- 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.

I Textbooks (9)

- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999.
- Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
- George E. Revesz. *Lambda-Calculus, Combinators and Functional Programming*. Cambridge University Press, 1988.
- 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 (10)

- Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002.
- Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998.
- Simon Thompson. *Haskell – The Craft of Functional Programming*. 2nd edition, Addison-Wesley/Pearson, 1999.
- Simon Thompson. *Haskell – The Craft of Functional Programming*. 3rd edition, Addison-Wesley/Pearson, 2011.
- Daniel J. Velleman. *How to Prove It. A Structured Approach*. Cambridge University Press, 1994.

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.
- 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)

- Jeremy Gibbons, Oege de Moor (Eds.). *The Fun of Programming*. Palgrave MacMillan, 2003.
- 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.

III Volumes (2)

- 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.
- 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)

- Hassan Ait-Kaci, Roger Nasr. *Integrating Logic and Functional Programming*. Lisp and Symbolic Computation 2(1):51-89, 1989.
- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
- Henry G. Baker. *Shallow Binding Makes Functional Arrays Fast*. ACM SIGPLAN Notices 26(8):145-147, 1991.
- 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.

IV Articles (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
- James R. Bitner, Edward M. Reingold. *Backtrack Programming Techniques*. Communications of the ACM 18(11):651-656, 1975.
- Matthias Blume, David McAllester. *Sound and Complete Models of Contracts*. Journal of Functional Programming 16(4-5):375-414, 2006.

IV Articles (3)

- 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.
- 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.

IV Articles (4)

- 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, 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.

IV Articles (5)

- 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.
- 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.

IV Articles (6)

- 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.
- 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.

IV Articles (7)

- 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.
- 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).

IV Articles (8)

- Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.
- Werner Damm, Bernhard Josko. *A Sound and Relatively* Complete Hoare-Logic for a Language with Higher Type Procedures*. Acta Informatica 20:59-101, 1983.
- 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.

IV Articles (9)

- 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
- Norbert Eisinger, Tim Geisler, Sven Panne. *Logic Implemented Functionally*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-V., LNCS 1292, 351-368, 1997.

IV Articles (10)

- Conal Elliot. *Functional Implementations of Continuous Modeled Animation*. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.
- Conal Elliot, Paul Hudak. *Functional Reactive Animation*. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.

IV Articles (11)

- 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 (12)

- 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 (13)

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 (14)

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 (15)

- Michael Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice*. Journal of Functional Programming 19&20:583-628, 1994.
- Michael Hanus (Ed.). *Curry: An Integrated Functional Logic Language*. Vers. 0.8.2, 2006. `www.curry-language.org/`
- Michael Hanus. *Multi-paradigm Declarative Languages*. In Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Springer-V., LNCS 4670, 45-75, 2007.

IV Articles (16)

- 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.
- Steve Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6(3):445-464, 1996.

IV Articles (17)

- Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
- Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.
- 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.

IV Articles (18)

- 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. *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.

IV Articles (19)

- 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.
- John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37:67-111, 2000.

IV Articles (20)

- 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 (21)

- 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, 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.
- Mark P. Jones. *Functional Thinking*. Lecture at the 6th International Summer School on Advanced Functional Programming, Boxmeer, The Netherlands, 2008.

IV Articles (22)

- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.
- 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.

IV Articles (23)

- 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 (24)

- 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.
- Rita Loogen, Yolanda Ortega-Mallén, Ricardo Pena-Mari. *Parallel Functional Programming in Eden*. Journal of Functional Programming 15(3):431-475, 2005.

IV Articles (25)

- 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.
- Lambert Meertens. *Calculating the Sieve of Eratosthenes*. Journal of Functional Programming 14(6):759-763, 2004.
- 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.

IV Articles (26)

- 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.
- 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.

IV Articles (27)

- 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.
- 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.

IV Articles (28)

- Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.
- 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 (29)

- 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 (30)

- 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 (31)

- 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 (32)

- 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 (33)

- 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 International Workshop on Implementation of Functional Languages (IFL 2000), Springer-V., LNCS 2011, 53-70, 2000.

IV Articles (34)

- Alastair Reid, John Peterson, Gregory 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.
- 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.

IV Articles (35)

- Colin Runciman, Matthew Naylor, Fredrik Lindblad. *SmallCheck and Lazy SmallCheck*. In Proceedings of the ACM SIGPLAN 2008 Workshop on Haskell (Haskell 2008), 37-48, 2008. (Available from `http://hackage.haskell.org`)
- 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.

IV Articles (36)

- Silvija Seres, Michael Spivey. *Embedding Prolog in Haskell*. In Proceedings of the 1999 Haskell Workshop (Haskell'99), 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.
- 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.

IV Articles (37)

- 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.
- Jay M. Spitzen, Karl M. Levitt, Lawrence Robinson. *An Example of Hierarchical Design and Proof*. Communications of the ACM 21(12):1064-1075, 1978.

IV Articles (38)

- 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.
- 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.

IV Articles (39)

- 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.
- 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.

IV Articles (40)

- Simon Thompson. *Proof*. In *Research Directions in Parallel Functional Programming*, Kevin Hammond, Greg Michaelson (Eds.), Springer-V., Chapter 4, 93-119, 1999.
- 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.

IV Articles (41)

- 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.
- 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.

IV Articles (42)

- 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.
- 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.

IV Articles (43)

- 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.
- Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 223-243, 2003.

IV Articles (44)

- 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

A
Mathematical Foundations

A.1

Sets and Relations

Sets and Relations (1)

Definition (A.1.1)

Let M be a set and R a relation on M, i.e. $R \subseteq M \times M$.

Then R is called

- reflexive iff $\forall\, m \in M.\ m\, R\, m$
- transitive iff $\forall\, m, n, p \in M.\ m\, R\, n \ \wedge\ n\, R\, p \ \Rightarrow\ m\, R\, p$
- anti-symmetric iff $\forall\, m, n \in M.\ m\, R\, n \wedge\ n\, R\, m \ \Rightarrow\ m = n$

Sets and Relations (2)

Related notions (though less important for us here):

Definition (A.1.2)

Let M be a set and $R \subseteq M \times M$ a relation on M. Then R is called

- symmetric iff $\forall\, m, n \in M.\ m\, R\, n \iff n\, R\, m$
- total iff $\forall\, m, n \in M.\ m\, R\, n \ \vee\ n\, R\, m$

A.2

Partially Ordered Sets

Partially Ordered Sets

Definition (A.2.1, Quasi-Order, Partial Order)

A relation R on M is called a

- quasi-order iff R is reflexive and transitive
- partial order iff R is reflexive, transitive, and anti-symmetric

For the sake of completeness we recall:

Definition (A.2.2, Equivalence Relation)

A relation R on M is called an

- equivalence relation iff R is reflexive, transitive, and symmetric

Note

...a partial order is an anti-symmetric quasi-order, an equivalence relation a symmetric quasi-order.

Remark: We here use terms like "partial order" as a short hand for the more accurate term "partially ordered set."

Bounds, least and greatest Elements

Definition (A.2.3, Bounds, least/greatest Elements)

Let $(Q, \sqsubseteq)$ be a quasi-order, let $q \in Q$ and $Q' \subseteq Q$.

Then q is called

- upper (lower) bound of Q', in signs: $Q' \sqsubseteq q$ ($q \sqsubseteq Q'$), if for all $q' \in Q'$ holds: $q' \sqsubseteq q$ ($q \sqsubseteq q'$)
- least upper (greatest lower) bound of Q', if q is an upper (lower) bound of Q' and for every other upper (lower) bound $\hat{q}$ of Q' holds: $q \sqsubseteq \hat{q}$ ($\hat{q} \sqsubseteq q$)
- greatest (least) element of Q, if holds: $Q \sqsubseteq q$ ($q \sqsubseteq Q$)

Existence and Uniqueness of Bounds

We have:

- Given a partial order, least upper and greatest lower bounds are uniquely determined, if they exist.
- Given existence (and thus uniqueness), the least upper (greatest lower) bound of a set $P' \subseteq P$ of the basic set of a partial order $(P, \sqsubseteq)$ is denoted by $\bigsqcup P'$ ($\bigsqcap P'$). These elements are also called supremum and infimum of P'.
- Analogously this holds for least and greatest elements. Given existence, these elements are usually denoted by $\bot$ and $\top$.

A.3

Lattices

Lattices and Complete Lattices

Definition (A.3.1, (Complete) Lattice)

Let $(P, \sqsubseteq)$ be a partial order.

Then $(P, \sqsubseteq)$ is called a

- lattice, if each finite subset P' of P contains a least upper and a greatest lower bound in P
- complete lattice, if each subset P' of P contains a least upper and a greatest lower bound in P

...(complete) lattices are special partial orders.

A.4

Complete Partially Ordered Sets

Complete Partial Orders

...a slightly weaker notion that, however, is often sufficient in computer science and thus often a more adequate notion:

Definition (A.4.1, Complete Partial Order)

Let $(P, \sqsubseteq)$ be a partial order.

Then $(P, \sqsubseteq)$ is called

- complete, or shorter a CPO (complete partial order), if each ascending chain $C \subseteq P$ has a least upper bound in P.

Remark

We have:

- A CPO $(C, \sqsubseteq)$ (more accurate would be: "chain-complete partially ordered set (CCPO)") has always a least element. This element is uniquely determined as the supremum of the empty chain and usually denoted by $\perp$: $\perp =_{df} \bigsqcup \emptyset$.

Chains

Definition (A.4.2, Chain)

Let $(P, \sqsubseteq)$ be a partial order.

A subset $C \subseteq P$ is called

- chain of P, if the elements of C are totally ordered. For $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}$ ($\{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\}$) we also speak more precisely of an ascending (descending) chain of P.

A chain C is called

- finite, if C is finite; infinite otherwise.

Finite Chains, finite Elements

Definition (A.4.3, Chain-finite)

A partial order $(P, \sqsubseteq)$ is called

- chain-finite (German: kettenendlich) iff P does not contain infinite chains

Definition (A.4.4, Finite Elements)

An element $p \in P$ is called

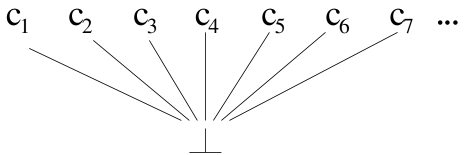
- finite iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ is free of infinite chains
- finite relative to $r \in P$ iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ does not contain infinite chains

(Standard) CPO Constructions (1)

Flat CPOs.

Let $(C, \sqsubseteq)$ be a CPO. Then $(C, \sqsubseteq)$ is called

- flat, if for all $c, d \in C$ holds: $c \sqsubseteq d \Leftrightarrow c = \bot \ \vee \ c = d$



(Standard) CPO Constructions (2)

Product construction.

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then

- the non-strict (direct) product $(\times P_i, \sqsubseteq)$ with
  - $(\times P_i, \sqsubseteq) = (P_1 \times P_2 \times \ldots \times P_n, \sqsubseteq)$ with $\forall (p_1, p_2, \ldots, p_n), (q_1, q_2, \ldots, q_n) \in \times P_i.\ (p_1, p_2, \ldots, p_n) \sqsubseteq (q_1, q_2, \ldots, q_n) \Leftrightarrow \forall i \in \{1, \ldots, n\}.\ p_i \sqsubseteq_i q_i$
- and the strict (direct) product (smash product) with
  - $(\bigotimes P_i, \sqsubseteq) = (P_1 \otimes P_2 \otimes \ldots \otimes P_n, \sqsubseteq)$, where $\sqsubseteq$ is defined as above under the additional constraint:

$$(p_1, p_2, \ldots, p_n) = \bot \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p_i = \bot_i$$

are CPOs, too.

(Standard) CPO Constructions (3)

Sum construction.

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ CPOs. Then

- the direct sum $(\bigoplus P_i, \sqsubseteq)$ with
  - $(\bigoplus P_i, \sqsubseteq) = (P_1 \dot\cup P_2 \dot\cup \ldots \dot\cup P_n, \sqsubseteq)$ disjoint union of P_i, $i \in \{1, \ldots, n\}$ and $\forall\, p, q \in \bigoplus P_i.\ p \sqsubseteq q \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q$

  is a CPO.

  Note: The least elements of $(P_i, \sqsubseteq_i)$, $i \in \{1, \ldots, n\}$, are usually identified, i.e., $\bot =_{df} \bot_i$, $i \in \{1, \ldots, n\}$

(Standard) CPO Constructions (4)

Function-space construction.

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be two CPOs and $[C \to D] =_{df} \{f : C \to D \mid f \text{ continuous}\}$ the set of continuous functions from C to D.

Then

- the continuous function space $([C \to D], \sqsubseteq)$ is a CPO where
  - $\forall f, g \in [C \to D].\ f \sqsubseteq g \iff \forall c \in C.\ f(c) \sqsubseteq_D g(c)$

Monotonic, Continuous Functions on CPOs

Definition (A.4.5, Monotonic, Continuous Function)

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be two CPOs and let $f : C \to D$ be a function from C to D.

Then f is called

- monotonic iff $\forall\, c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$
  (Preservation of the ordering of elements)
- continuous iff $\forall\, C' \subseteq C.\ f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$
  (Preservation of least upper bounds)

Properties

Using the notations introduced before, we have:

Lemma (A.4.6)

f is monotonic iff $\forall\, C' \subseteq C.\ f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$

Corollary (A.4.7)

A continuous function is always monotonic, i.e. f continuous implies f monotonic.

Inflationary Functions on CPOs

Definition (A.4.8, Inflationary Function)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a function on C. Then f is called

- inflationary (increasing) iff $\forall\, c \in C.\ c \sqsubseteq f(c)$

A.5

Fixed Point Theorem

Least and Greatest Fixed Points (1)

Definition (A.5.1, (Least/Greatest) Fixed Point)

Let $(C, \sqsubseteq)$ be a CPO, $f : C \rightarrow C$ be a function on C and let c be an element of C, i.e., $c \in C$.

Then c is called

- fixed point of f iff $f(c) = c$

A fixed point c of f is called

- least fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$
- greatest fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow d \sqsubseteq c$

Notation:

- The least resp. greatest fixed point of a function f is usually denoted by μf resp. νf.

Least and Greatest Fixed Points (2)

Definition (A.5.2, Conditional Fixed Point)

Let $(C, \sqsubseteq)$ be a CPO, $f : C \to C$ be a function on C and let $d, c_d \in C$.

Then c_d is called

- conditional (German: bedingter) least fixed point of f wrt d iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e. for all other fixed points x of f with $d \sqsubseteq x$ holds: $c_d \sqsubseteq x$.

Fixed Point Theorem

Theorem (A.5.3, Knaster/Tarski, Kleene)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a continuous function on C.

Then f has a least fixed point μf, which equals the least upper bound of the chain (so-called Kleene-Chain) $\{\bot, f(\bot), f^2(\bot), \ldots\}$, i.e.

$$\mu f = \bigsqcup_{i \in I\!N_0} f^i(\bot) = \bigsqcup \{\bot, f(\bot), f^2(\bot), \ldots\}$$

Proof of Fixed Point Theorem A.5.3 (1)

We have to prove:

μf

1. exists
2. is a fixed point
3. is the least fixed point

of f.

Proof of Fixed Point Theorem A.5.3 (2)

1. Existence

- It holds $f^0 \perp = \perp$ and $\perp \sqsubseteq c$ for all $c \in C$.
- By means of (natural) induction we can show: $f^n \perp \sqsubseteq f^n c$ for all $c \in C$.
- Thus we have $f^n \perp \sqsubseteq f^m \perp$ for all n, m with $n \leq m$. Hence, $\{f^n \perp \mid n \geq 0\}$ is a (non-finite) chain of C.
- The existence of $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is thus an immediate consequence of the CPO properties of $(C, \sqsubseteq)$.

Proof of Fixed Point Theorem A.5.3 (3)

2. Fixed point property

$$
\begin{array}{rcl}
 & & f(\bigsqcup_{i\in\mathbb{N}_0} f^i(\bot)) \\
(f \text{ continuous}) & = & \bigsqcup_{i\in\mathbb{N}_0} f(f^i\bot) \\
 & = & \bigsqcup_{i\in\mathbb{N}_1} f^i\bot \\
(K \text{ chain} \Rightarrow \bigsqcup K = \bot \sqcup \bigsqcup K) & = & (\bigsqcup_{i\in\mathbb{N}_1} f^i\bot) \sqcup \bot \\
(f^0\bot = \bot) & = & \bigsqcup_{i\in\mathbb{N}_0} f^i\bot
\end{array}
$$

3. Least fixed point

- Let c be an arbitrarily chosen fixed point of f. Then we have $\perp \sqsubseteq c$, and hence also $f^n \perp \sqsubseteq f^n c$ for all $n \geq 0$.
- Thus, we have $f^n \perp \sqsubseteq c$ because of our choice of c as fixed point of f.
- Thus, we also have that c is an upper bound of $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$.
- Since $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is the least upper bound of this chain by definition, we obtain as desired $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c$.

$\square$

Conditional Fixed Points

Theorem (A.5.4, Conditional Fixed Points)

Let $(C, \sqsubseteq)$ be a CPO, let $f : C \to C$ be a continuous, inflationary function on C, and let $d \in C$.

Then f has a unique conditional fixed point μf_d. This fixed point equals the least upper bound of the chain $\{d, f(d), f^2(d), \ldots\}$, d.h.

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}$$

Finite Fixed Points

Theorem (A.5.5, Finite Fixed Points)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a continuous function on C.

Then we have: If two elements in a row occurring in the Kleene-chain of f are equal, e.g. $f^i(\bot) = f^{i+1}(\bot)$, then we have: $\mu f = f^i(\bot)$.

Existence of Finite Fixed Points

Sufficient conditions for the existence of finite fixed points e.g. are

- Finiteness of domain and range of f
- f is of the form $f(c) = c \sqcup g(c)$ for monotone g on some chain-complete domain

A.6

Cones and Ideals

Cones und Ideals

Definition (A.6.1, Directed Set, Cone, Ideal)

Let $(P, \sqsubseteq)$ be a partial order and Q be a subset of P, i.e., $Q \subseteq P$.

Then Q is called

- directed set (German: gerichtet (gerichtete Menge)), if each *finite* subset $R \subseteq Q$ has a supremum in Q, i.e. $\exists\, q \in Q.\ q = \bigsqcup R$
- cone (German: Kegel), if Q is downward closed, i.e. $\forall\, q \in Q\ \forall\, p \in P.\ p \sqsubseteq q \Rightarrow p \in Q$
- ideal (German: Ideal), if Q is a directed cone, i.e. if Q is downward closed and each finite subset has a supremum in Q.

Note: If Q is a directed set, then, we have because of $\emptyset \subseteq Q$ also $\bigsqcup \emptyset = \bot \in Q$ and thus $Q \neq \emptyset$.

Completion of Ideals

Theorem (A.6.2, Completion of Ideals)

Let $(P, \sqsubseteq)$ be a partial order and let I_P be the set of all ideals of P. Then we have:

- *$(I_P, \subseteq)$ is a CPO.*

Induced "completion:"

- Identifying each element $p \in P$ with its corresponding ideal $I_p =_{df} \{q \mid q \sqsubseteq p\}$ yields an embedding of P into I_P with $p \sqsubseteq q \;\Leftrightarrow\; I_P \subseteq I_Q$

Corollary (A.6.3, Extensibility of Functions)

Let $(P, \sqsubseteq_P)$ be a partial order and let $(C, \sqsubseteq_C)$ be a CPO. Then we have: All monotonic functions $f : P \to C$ can be extended to a uniquely determined continuous function $\hat{f} : I_P \to C$.

Summing up

The preceding result implies:

- Streams constitute a CPO.
- Recursive equations and functions on streams are well-defined
- The application of a function to the finite prefixes of a stream yields the chain of approximations of the application of the function to the stream itself; it is thus correct.

Appendix A: Further Reading (1)

- 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)
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 4, Denotational Semantics)
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 5, Denotational Semantics)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten)

Appendix A: Further Reading (2)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 8, Reasoning about Programs; Chapter 17.9, Proof revisited)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about Programs; Chapter 17.9, Proof revisited)