Fortgeschrittene funktionale Programmierung

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

(Stand: 29.04.2020)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Lecture 2

Part IV: Advanced Language Concepts

- Chapter 7: Functional Arrays
- Chapter 8: Abstract Data Types

Part II: Programming Principles

- Chapter 3: Programming with Higher-Order Functions: Algorithm Patterns

Outline in more Detail (1)

Part IV: Advanced Language Concepts

Outline in more Detail (2)

Part II: Programming Principles

Chapter 7

Functional Arrays

Chapter 7.1

Motivation

Imperative Arrays

...appealing:

+ Values of an array can be accessed or updated in constant time.
+ The update operation does not need extra space.
+ There is no need for chaining the array elements with pointers as they can be stored in contiguous memory locations.

...distracting:

– The size is fixed (defined and fixed at declaration time).

Functional Lists

...appealing:

+ The size is not fixed. Lists can get, can be arbitrarily long, conceptually even infinitely long.

...distracting:

– Lists do not enjoy the set of favorable properties of imperative arrays; most disturbing, values of a list can not be accessed or updated in constant time:

  Accessing the ith element of a list (using `(!!)`) takes a number of steps proportional to i.

Functional Arrays

...shall complement functional lists and be designed and implemented to get as close as possible to the favorable properties of imperative arrays, i.e., functional arrays shall be:

...appealing because:

+ Accessing the ith element of an array (using `(!)`) shall take a constant number of steps, regardless of i.

...while accepting the distracting limitation applying to imperative arrays, too:

– The size is fixed (defined and fixed at creation time).

Note: Functional Arrays

...are not supported by the standard prelude of Haskell but by several specialized libraries like:

- `Data.Array` ($\rightsquigarrow$ `import Data.Array`)
- `Data.Array.IArray` ($\rightsquigarrow$ `import Data.Array.IArray`)
- `Data.Array.Diff` ($\rightsquigarrow$ `import Data.Array.Diff`)

providing different kinds and implementations of functional arrays:

- Static (or: immutable) arrays (w/out destructive update)
- Dynamic (or: mutable) arrays (w/ destructive update)

Nonetheless, how to implement functional arrays

- most adequately is a topic of ongoing research.

Consequently, libraries evolve, disappear, are replaced over time requiring to stay tuned for updates on the Haskell homepage...

Chapter 7.2

Functional Arrays

Chapter 7.2.1

Static Arrays

The Library `Array`

- `Array` (⇝ `import Array`)

...supports static arrays and provides three functions for creating static arrays:

1. `array` *bounds list_of_associations* (1st mechanism)
2. `listArray` *bounds list_of_values* (2nd mechanism)
3. `accumArray` *f init bounds list_of_associations* (3rd mechanism)

Important: The type class `Ix`, whose instance types are (mainly) used as index types of arrays:

```
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: 1st Mechanism

...using the function `array`, the most basic means:

- `array :: Ix a => (a,a) -> [(a,b)] -> Array a b`
  `array` *bounds list_of_associations*

where

- `a`: the index type of the array; `b`: its entry type.
- *bounds*: a pair of expressions specifying the smallest and the largest array index.

  Example: The expression pair bounds
  a) `(0,4)` and b) `((1,1),(10,10))` specify a
  a) zero-origin vector of length five
  b) one-origin 10 by 10 matrix, respectively.

  Note: *bounds* can be given by any valid expression.
- *list_of_associations*: a list of pairs `(i,x)`, so-called associations, specifying that the array entry at index position `i` has value `x`.

Examples: array at Work

Let a′, f n, m be the expressions:

```
a'   = array (1,4) [(3,'c'),(2,' a'),(1,'f'),(4,'e')]
f n = array (0,n) [(i,i*i) | i <- [0..n]]
m   = array ((1,1),(2,3))
              [((i,j),(i*j)) | i <- [1..2], j <- [1..3]]
```

The types of these expressions are:

```
a' :: Array Int Char
f  :: Int -> Array Int Int
m  :: Array (Int,Int) Int
```

Their values are:

```
a'   ->> array (1,4) [(1,'f'),(2, 'a'),(3,'c'),(4,'e')]
f 3 ->> array (0,3) [(0,0),(1,1),(2,4),(3,9)]
m    ->> array ((1,1),(2,3)) [((1,1),1),((1,2),2),
                               ((1,3),3),((2,1),2),
                               ((2,2),4),((2,3),6)]
```

Note

If any specified index of an array is out of bounds

- the whole array is undefined.

  I.e.: Function `array` is strict in bounds.

If two associations in an association list have the same index

- the array entry at that index is undefined.

  I.e.: Function `array` is non-strict (or: lazy) in values.

...arrays can thus contain 'undefined' entries.

Example: Arrays at Work

Computing Fibonacci numbers:

```
fibs n = a
 where a = array (1,n) ([(1,0), (2,1)] ++
                        [(i, a!(i-1) + a!(i-2))
                          | i <- [3..n]])
```

Applications:

```
fibs  3 ->> array (1,3) [(1,0),(2,1),(3,1)]

fibs  5 ->> array (1,5) [(1,0),(2,1),(3,1),
                         (4,2),(5,3)]

fibs 12 ->> array (1,12) [(1,0),(2,1),(3,1),
                          (4,2),(5,3),(6,5),
                          (7,8),(8,13),(9,21),
                          (10,34),(11,55),(12,89)]
```

The Array Access Function (!)

...the counterpart of the list access function `(!!)` for arrays:

```
(!) :: Ix a => Array a b -> a -> b
```

`(!)` returns the value `v :: b` at index position `i :: a`.

Recall: The index type must be a member of the type class `Ix`, which foresees maps for typical index operations.

The Array Access Function (!) at Work

Computing Fibonacci numbers:

```
fibs n = a where
              a = array (1,n)
               ([(1,0),(2,1)] ++ [(i,a!(i-1)+,a!(i-2))
                                   | i <- [3..n]])
```

Applications of (!):

```
fibs 5!5     ->> 3
fibs 10!10   ->> 34
fibs 100!10  ->> 34 -- Thanks to lazy evaluation,
                    -- the computation stops at
                    -- fibs 10!10
fibs 50!50   ->> 7.778.742.049
fibs 100!100 ->> 218.922.995.834.555.169.026
fibs 5!10    ->> Program error: Ix.index: index
                 out of range
```

Note: Local Declarations for Performance (1)

...the `where`-clause in the definition of `fibs` defining `a` locally is crucial for performance as it

- avoids the creation of new arrays during computation.

For illustration, compare the definitions of `fibs` and `xfibs`, where `a` (of a slightly different type) is globally defined:

```
fibs n = a where
            a = array (1,n)
             ([(1,0),(2,1)] ++ [(i,a!(i-1)+,a!(i-2))
                                       | i <- [3..n]])

xfibs n = a n
a n     = array (1,n) ([(1,0),(2,1)] ++
                        [(i,a n!(i-1) + a n!(i-2))
                          | i <- [3..n] ])
```

Note: Local Declarations for Performance (2)

While

```
xfibs  3 ->> array (1,3) [(1,0),(2,1),(3,1)]
xfibs  5 ->> array (1,5) [(1,0),(2,1),(3,1),(4,2),(5,3)]
xfibs 12 ->> array (1,12) [(1,0),(2,1),(3,1),
                            (4,2),(5,3),(6,5),
                            (7,8),(8,13),(9,21),
                            (10,34),(11,55),(12,89)]
xfibs 5!5   ->> 3
xfibs 10!10 ->> 34
xfibs 25!20 ->> 4.181  -- thanks to lazy evaluation
                       -- the computation stops asap
```

works well for small arguments, the call:

```
xfibs 25!25 ->> ...takes too long to be feasible!
```

Overall: Though correct, evaluating `xfibs n` is most inefficient due to creating new arrays during the evaluation.

Creating Static Arrays: 2nd Mechanism

...using the function `listArray`, a more sophisticated means:

- `listArray :: (Ix a) => (a,a) -> [b] -> Array a b`
  `listArray` *bounds* *list_of_values*

where

- *bounds*: specifies the values of the smallest and the largest index.
- *list_of_values*: specifies the values of the array elements in terms of a list.

Note: `listArray` is especially useful for the frequent case where

- where an array is constructed from a list of values given in index order.

Example: listArray at Work

```
a'' :: Array Int Char
a'' = listArray (1,8) "fun prog"
```

Evaluating a'' yields:

```
a'' ->> array (1,8) [(1,'f'),(2,'u'),(3,'n'),(4,' '),
                     (5,'p'),(6,'r'),(7,'o'),(8,'g')]
```

Creating Static Arrays: 3rd Mechanism

...using the function `accumArray`, the most powerful means:

- ```
  accumArray :: (Ix a) => (b -> c -> b) -> b
                 -> (a,a) -> [(a,c)] -> Array a b
  ```
  `accumArray` *f init bounds list_of_associations*

where

- *f*: specifies an accumulation function.
- *init*: specifies the (default) value the entries of the array shall be initialized with.
- *bounds*: specifies the values of the smallest and the largest index.
- *list_of_associations*: specifies the values of the array in terms of an association list.

Note: `accumArray` does not require that the indices occurring in list_of_associations are pairwise disjoint: Values of 'conflicting' indices are accumulated via *f*.
  ```

Example: accumArray at Work (1)

...a histogram function defined with accumArray:

```
histogram :: (Ix a, Num b) =>
                          (a,a) -> [a] -> Array a b
histogram bounds vs =
 accumArray (+) 0 bounds [(i,1) | i <- vs]
```

Applications:

```
histogram (1,5) [4,1,4,3,2,5,5,1,2,1,3,4,2,1,1,3,2,1]
 ->> array (1,5) [(1,6),(2,4),(3,3),(4,3),(5,2)]

histogram (-1,4) [1,3,1,1,3,1,1,3,1]
 ->> array (-1,4) [(-1,0),(0,0),(1,6),(2,0),(3,3),(4,0)]

histogram (1,3) [5,3,1,3,4,2,(-4),1,1,3,2,1,5,(-9)]
 ->> array
     Program error: Ix.index: index out of range
```

Example: accumArray at Work (2)

...a prime number test defined with accumArray:

```
primes :: Int -> Array Int Bool
primes n =
   accumArray (\e e′ -> False) True (2,n) l
    where l = concat [map (flip (,) ())
                   (takeWhile (<=n) [k*i|k<-[2..]])
                                      | i<-[2..n ‘div‘ 2]]
```

Applications:

```
(primes 100)!1   ->> Program error: Ix.index: index
                     out of range
(primes 100)!2   ->> True
(primes 100)!4   ->> False
(primes 100)!71  ->> True
(primes 100)!100 ->> False
(primes 100)!101 ->> Program error: Ix.index: index
                     out of range
```

More Pre-Defined Operations on Arrays (1)

..pre-defined array operations:

```
– (!)      :: (Ix a) => Array a b -> a -> b
– bounds   :: (Ix a) => Array a b -> (a,a)
– indices  :: (Ix a) => Array a b -> [a]
– elems    :: (Ix a) => Array a b -> [b]
– assocs   :: (Ix a) => Array a b -> [(a,b)]
– (//)     :: (Ix a) => Array a b -> [(a,b)]
                                      -> Array a b
– ...
```

More Pre-Defined Operations on Arrays (2)

Informally:

- `(!)`: array subscripting, yields the *i*th element of an array.
- `bounds`: yields the smallest and largest index of an array.
- `indices`: yields a list of the indices of an array.
- `elems`: yields a list of the elements/values of an array.
- `assocs`: yields a list of index/value pairs of the elements of an array, i.e., the list of associations of an array.
- `(//)`: array updating – `(//)` takes an array (left argument) and a list of associations (right argument) and returns a new array, which is identical to the argument array except for the values of elements occurring in the argument list of associations.

  Note: `(//)` generates a modified copy of the argument array; it does not perform a destructive update!
- ...

Example: More Array Operations at Work (1)

Applications (w/ pre-defined functions on arrays):

```
elems (primes 10)
 ->> [True,True,False,True,False,True,False,False,False]

assocs (primes 10)
 ->> [(2,True),(3,True),(4,False),(5,True),(6,False),
      (7,True),(8,False),(9,False),(10,False)]

yieldPrimes (assocs (primes 100))
 ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
      59,61,67,71,73,79,83,89,97]
```

where

```
yieldPrimes :: [(a,Bool)] -> [a]
yieldPrimes [] = []
yieldPrimes ((v,w):t)
 | w         = v : yieldPrimes t
 | otherwise = yieldPrimes t
```

Example: More Array Operations at Work (2)

Let:

```
m = array ((1,1),(2,3)) [((i,j),i*j) | i <- [1..2],
                                        j <- [1..3]]
                                :: Array (Int,Int) Int

m ->> array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
                           ((2,1),2),((2,2),4),((2,3),6)]

m!(1,2) ->> 2, m!(2,2) ->> 4, m!(2,3) ->> 6
```

Applications of array operations:

```
bounds   m ->> ((1,1),(2,3))
indices  m ->> [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
elems    m ->> [1,2,3,2,4,6]
assocs   m ->> [((1,1),1),((1,2),2),((1,3),3),
               ((2,1),2), ((2,2),4), ((2,3),6)]

m // [((1,1),4), ((2,2),8)]
 ->> array ((1,1),(2,3)) [((1,1),4),((1,2),2),((1,3),3),
                          ((2,1),2),((2,2),8),((2,3),6)]
```

Example: More Array Operations at Work (3)

...illustrating the update operation (//) by means of modifying the histogram function:

```
histogram (lower,upper) xs
 = updHist (array (lower,upper)
                  [(i,0) | i <- [lower..upper]])
            xs

updHist a []     = a
updHist a (x:xs) = updHist (a // [(x, (a!x + 1))]) xs
```

Application:

```
histogram (0,9) [3,1,4,1,5,9,2]
 ->> array (0,9) [(0,0),(1,2),(2,1),(3,1),(4,1),
                  (5,1),(6,0),(7,0),(8,0),(9,1)]
```

Updating Arrays: accum complementing (//)

...accum, another pre-defined operation on arrays:

- `accum :: (Ix a) => (b -> c -> b) -> Array a b`
  `                              -> [(a,c)] -> Array a b`

  accum *f a list_of_associations*

...instead of replacing previously stored values as (//) does, accum accumulates values referring to the same index using f.

Example:

```
accum (+) m [((1,1),4), ((2,2),8)]    -- m as before
 ->> array ((1,1),(2,3))
          [((1,1),5),((1,2),2),((1,3),3),
           ((2,1),2),((,2,2),12),((2,3),6)]
```

Note: The result of accum is a new matrix, which is identical to m except for the entries at positions (1,1) and (2,2) to whose values 1 and 4, 4 and 8 have been added, respectively.

Higher-Order Functions on Arrays

...can be defined just as on lists, e.g.:

```
amap :: (b -> c) -> Array a b -> Array a c
```

Example: The call

```
amap (\x -> x*10) a
```

yields a new array where all elements of a are multiplied by 10.

User-defined Higher-Order Array Functions

The functions row and col return a row and a column of a matrix, respectively:

```
row :: (Ix a,Ix b) =>
                a -> Array (a,b) c -> Array b c
row i m = ixmap (l',u') (\j->(i,j)) m
            where ((l,l'),(u,u')) = bounds m

col :: (Ix a,Ix b) =>
                a -> Array (b,a) c -> Array b c
col j m = ixmap (l,u) (\i->(i,j)) m
            where ((l,l'),(u,u')) = bounds m
```

where

```
ixmap :: (Ix a, Ix b) => (a,a) -> (a -> b)
                           -> Array b c -> Array a c
ixmap b f a = array b [(k,a!f k) | k <- range b]
```

Examples: row, col at Work

...where m is assumed to be as before:

```
row 1 m ->> array (1,3) [(1,1),(2,2),(3,3)]
row 2 m ->> array (1,3) [(1,2),(2,4),(3,6)]
row 3 m ->> array (1,3) [(1,
            Program error: Ix.index: index out of
            range

col 1 m ->> array (1,2) [(1,1),(2,2)]
col 2 m ->> array (1,2) [(1,2),(2,4)]
col 3 m ->> array (1,2) [(1,3),(2,6)]
col 4 m ->> array (1,2) [(1,
            Program error: Ix.index: index out of
            range
```

Chapter 7.2.2

Dynamic Arrays

The Library `Data.Array.Diff`

- `Data.Array.Diff` ($\rightsquigarrow$ `import Data.Array.Diff`)

...supports dynamic (or: mutable) arrays.

Compared to the library `Data.Array`, the type:

– `DiffArray` (for dynamic arrays)

replaces the type

– `Array` (for static arrays)

...everything else behaves analogously.*)

*) `Data.Array.Diff` is no longer maintained; `Data.Array.IO` can be considered a substitute but offers a different monadic-based interface.

Chapter 7.3

Summary

Summing up (1)

Static (Immutable) Arrays

- Access operator (!): Each array element is accessible in constant time.
- Update operator (//): Not a destructive update; instead: an identical copy of the argument array is created except of those elements being 'updated.' Updates thus do not take constant time.

Dynamic (Mutable) Arrays

- Update operator (//): Destructive update; update operations take constant time per index.
- Access operator (!): Access to array elements may sometimes take longer as for static arrays.

Summing up (2)

Updates

- can often completely be avoided by smartly written recursive array constructions (cp. the prime number test in Chapter 7.2.1).

Dynamic arrays

- should only be used if constant time updates are crucial for the application.

For an extended example showing

– arrays at work

refer to Chapter 18.2 dealing with an imperative robot language for controlling robot actions.

Chapter 7.4

References, Further Reading

Chapter 7: Basic Reading (1)

Basics, Fundamentals of Functional Arrays

Klaus E. Grue. *Arrays in Pure Functional Programming Languages*. International Journal on Lisp and Symbolic Computation 2:105-113, Kluwer Academic Publishers, 1989.

Textbook Representations on Functional Arrays

Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 10.5, Mutable arrays; Chapter 10.6, Immutable arrays)

Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Chapter 10.1, Arrays)

Chapter 7: Basic Reading (2)

- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.6, Arrays)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 2.7, Arrays; Chapter 4.3, Arrays)

Functional Arrays in Haskell'98

- 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.

Chapter 7: Selected Further Reading (1)

(Towards) Fast Implementations of Functional Arrays

- Henry G. Baker. *Shallow Binding Makes Functional Arrays Fast*. ACM SIGPLAN Notices 26(8):145-147, 1991.
- 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.
- John Hughes. *An Efficient Implementation of Purely Functional Arrays*. Technical Report, Programming Methodology Group, Chalmers University of Technology, 1985.

Chapter 7: Selected Further Reading (2)

Miscellaneous Aspects of Functional Arrays

- 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.
- 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)

And now on something completely different

...towards Abstract Data Types, towards Chapter 8.

Data Type Description Modes

Consider:

- A tree is either a leaf carrying a string value, or it is a branch carrying an integer value and a right and a left tree, so-called subtrees.
- The function
  - emptystack creates a new stack, the so-called empty stack, which does not contain any entry.
  - push adds a new entry to a stack.
  - pop removes the entry of a stack, which has most recently been added to it; if there is none, it fails.
  - top yields the entry of the stack, which has most recently been added to it; if there is none, it fails.
  - isempty checks if a stack contains an entry.

  There is no other way to create, access, and manipulate stack values as by means of these functions.

Exercise

Considering and comparing the two descriptions:

1. What do they tell us
2. What do they not tell us

about trees and stacks, about tree values and stack values?

Can we, based on these descriptions, provide an implementation of

1. trees
2. stacks

in, e.g., Haskell?

Exercise (cont'd)

What is about the below implementations of trees and stacks?

```
data Tree = Leaf String
          | Branch Int Tree Tree

type Stack a = [a]

emptystack = []

push x xs  = (x:xs)

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

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

isempty [] = True
isempty _  = False
```

1. Are they faithful implementations of the descriptions?
2. Can they faithfully be derived from the descriptions?

The Description of Trees

...tells us

- everything about the (concrete) values of trees, about what they look like.
- nothing about functions which allow us to create, access and manipulate tree values.

...we call trees a concrete data type description.

Recall:

A tree is either a leaf carrying a string value, or it is a branch carrying an integer value and a right and a left tree, so-called subtrees.

The Description of Stacks

...tells us

- everything about the functions at our disposal for creating, accessing, and manipulating values of the data type.
- nothing about the (concrete) values of the data type, about what they look like.

...we call stacks an abstract data type description.

Recall: Calling function

- emptystack creates a new stack, the so-called empty stack, which does not contain any entry.
- push adds a new entry to a stack.
- pop removes the entry of a stack, which has most re- cently been added to it; if there is none, it fails.
- top yields the entry of the stack, which has most recently been added to it; if there is none, it fails..
- isempty checks if a stack contains an entry.

There is no other way to create, access, and manipulate stack values than by means of these functions.

Summing up

A concrete data type description (like trees) tells us

- everything about the values of the data type, about what they look like.
- nothing about functions allowing us to create, access, and manipulate values of the data type.

An abstract data type description like (stacks) tells us

- everything about the functions allowing us to create, access, and manipulate values of the data type.
- nothing about the kind of values of the data type, about what they look like.

Exercise

Natural language is ambiguous, suggestive, open and inviting to (over-) interpretation. E.g., the description of stacks:

- Calling function
  - emptystack creates a new stack, the so-called empty stack, which does not contain any entry.
  - push adds a new entry to a stack.
  - pop removes the entry of a stack, which has most recently been added to it; if there is none, it fails.
  - top yields the entry of the stack, which has most recently been added to it; if there is none, it fails.
  - isempty checks if a stack contains an entry.

  There is no other way to create, access, and manipulate stack values than by means of these functions.

tells us actually only

- almost everything about functions which allow us to create, access, and manipulate stack values.

Lecture 2
Detailed Outline
Chap. 7
Towards ADTs
Chap. 8
Chap. 3
Final Note

Exercise (cont'd)

What information is missing in the description of stacks or (over-) interpreted to justify an implementation of stacks as shown below?

```
type Stack a = [a]
emptystack = []
push x xs  = (x:xs)
pop []     = error "Stack is empty"
pop (_:xs) = xs
top []     = error "Stack is empty"
top (x:_)  = x
isempty [] = True
isempty _  = False
```

In the following Chapter 8

...we will show (one way of)

- how to provide precise abstract data type (ADT) descriptions by decomposing their description into a
  - user-visible specification.
  - user-invisible implementation.
  - verification obligations for specifier and implementer.

Moreover, we will show:

- What benefits and advantages ADT definitions provide.
- How ADT definitions can be be realized in Haskell.

Chapter 8

Abstract Data Types

Chapter 8.1

Motivation

Why Abstract Data Types?

...by introducing a level of indirection between specification and implementation of a data type, we achieve:

- Separation of concerns: Separation of specification (interface and behaviour specification) and implementation of a data type (in terms of a CDT and CDT operations matching the ADT operations).
- Information hiding: No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.
- Security: CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated using the ADT operations implemented by their CDT counterparts.

Defining and Implementing an ADT

...is technically a three-stage approach of specification, implementation, and verification:

- Specification (user-visible)
  - Interface Specification: Signatures of ADT operations
  - Behaviour Specification: Laws for ADT operations
- Implementation (user-invisible)
  - Implementing the ADT values in terms of a CDT
  - Implementing the ADT operations as CDT operations
- Verification
  - Specification: Proving that the ADT laws are consistent and complete (proof obligation of the ADT specificator)
  - Implementation: Proving that the implemented CDT operations are sound, i.e., satisfy the ADT laws (proof obligation of the CDT implementor)

Benefits of Abstract Data Type Definitions

...supporting programming-in-the large:

- ADTs enable modular program development by separating the responsibilities for specifying and implementing a data type and the operations associated with it.

...supporting reusability and maintainability:

- If non-functional requirements for an ADT implementation change or evolve over time, a current CDT implementation of the ADT and its operations can easily be replaced by a new one fitting better to the new requirements as long as the new CDT implementation satisfies the interface and behaviour specification of the ADT.

In the following

...we demonstrate how ADTs can be defined and implemented in Haskell considering:

- Stacks
- Queues
- Priority Queues
- Tables

The Challenge:

- ADTs are not a first-class citizen in Haskell.
- Therefore, we have to pragmatically make use of Haskell features allowing us to achieve the constituting properties of ADTs of information hiding, of separating their user-visible specification from their user-invisible implementation as good as possible.

Chapter 8.2

Stacks

Interface Specification

...of the ADT stack, named Stack (user-visible):

```
module Stack (Stack,empty,is_empty,push,pop,top)
                                                where

-- Interface Spec.: Signatures of stack operations
empty    :: Stack a
is_empty :: Stack a -> Bool
push     :: a -> Stack a -> Stack a
pop      :: Stack a -> Stack a
top      :: Stack a -> a

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

Note, the laws must be chosen to enforce a last-in/first-out (LIFO) behaviour of stacks; any implementation of stacks must ensure these laws.

Behaviour Specification

...of the stack operations of the ADT stack (user-visible):

```
Behaviour Spec.: Laws for stack operations

1) is_empty empty        == True
2) is_empty (push v s) == False
3) top empty             == undef
4) top (push v s)        == v
5) pop empty             == undef
6) pop (push v s)        == s
```

Homework: Prove that the above laws enforce a last-in/first-out (LIFO) behaviour of stacks.

Implementation A

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

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

empty           = Empty

is_empty Empty  = True
is_empty _      = False

push x s        = Stk x s

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

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

Implementation B

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

```
newtype Stack a    = Stk [a]

empty              = Stk []

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

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

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

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

“Implementation” C

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

```
type Stack a = [a]

empty          = []

is_empty []    = True
is_empty _     = False

push x xs      = (x:xs)

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

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

Verification

Specifier and implementer of the ADT stack can prove, respectively:

Lemma 8.2.1 (Consistency, Completeness)

The 6 laws of the behaviour specification of the ADT stack are consistent and complete.

Lemma 8.2.2 (Soundness)

The implementations A and B (and C) satisfy the 6 laws of the behaviour specification of the ADT stack.

A Critical Note on “Implementation” C

…of stacks as an

– alias type of predefined lists: `type Stack a = [a]`

Obvious (but actually only apparent) benefit of implementing stacks as predefined lists:

– Even less conceptual overhead than for stacks implemented as a new type `newtype Stack a = Stk [a]` where the constructor `Stk` needs to be handled by the implementations of the stack operations.

But

Security is broken and lost!

- All predefined operations on lists are available on stacks (not just the 5 ADT operations of stack).

Worse

- Many of the predefined operations on lists (reversal, element picking, etc.) are not even meaningful for stacks.
- Even hiding the implementation in a module can not prevent the application of such meaningless operations to stacks but requires to explicitly abstain from them.

Hence

- "Implementation" C violates the spirit of an ADT implementation and should not be considered a reasonable and valid implementation of the ADT stack.

Chapter 8.3

Queues

Interface Specification

...of the ADT queue, named Queue (user-visible):

```
module Queue (Queue,emptyQ,is_EmptyQ,
                    enQ,deQ,frontQ) where

-- Interface Spec.: Signatures of queue operations
emptyQ    :: Queue a
is_emptyQ :: Queue a -> Bool
enQ       :: a -> Queue a -> Queue a
deQ       :: Queue a -> Queue a
frontQ    :: Queue a -> a

-- Behaviour Spec.: Laws for queue operations
Laws (1) thru (6)
```

Note, the laws must be chosen to enforce a first-in/first-out (FIFO) behaviour of queues; any implementation of queues must ensure these laws.

Behaviour Specification

...of the queue operations of the ADT queue (user-visible):

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

1) is_emptyQ emptyQ     == True
2) is_emptyQ (enQ v q) == False
3) frontQ emptyQ        == undef
4) frontQ (enQ v q)     == if is_emptyQ q
                             then v
                             else frontQ q
5) deQ emptyQ           == undef
6) deQ (enQ v q)        == if is_emptyQ q
                             then emptyQ
                             else enQ ((deQ q) v)
```

Homework: Prove that the above laws enforce a first-in/first-out (FIFO) behaviour of queues.

Implementation A

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

```
newtype Queue a  = Q [a]

emptyQ            = Q []

is_emptyQ (Q []) = True
is_emptyQ _       = False

enQ x (Q q)       = Q (q ++ [x])

deQ (Q [])        = error "Queue is empty"
deQ (Q (_:xs))    = Q xs

frontQ (Q [])     = error "Queue is empty"
frontQ (Q (x:_)) = x
```

Implementation B

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

```
newtype Queue a          = Q ([a],[a])
```

front *rear* (*in reverse order of the queue*)

```
emptyQ                   = Q ([],[])
is_emptyQ (Q ([],[]))    = True
is_emptyQ _              = False
enQ x (Q ([],[]))        = Q ([x],[])
enQ y (Q (xs,ys))        = Q (xs,y:ys)
deQ (Q ([],[]))          = error "Queue is empty"
deQ (Q ([],ys))          = Q (tail(reverse ys),[])
deQ (Q (x:xs,ys)          = Q (xs,ys)
frontQ (Q ([],[]))       = error "Queue is empty"
frontQ (Q ([],ys)        = last ys
frontQ (Q (x:xs,ys)      = x
```

Verification

Specifier and implementer of the ADT queue can prove, respectively:

Lemma 8.3.1 (Consistency, Completeness)

The 6 laws of the of the behaviour specification of the ADT queue are consistent and complete.

Lemma 8.3.2 (Soundness)

The implementations A and B satisfy the 6 laws of the behaviour specification of the ADT queue.

Exercise 8.3.3

Implementation B of the ADT queue is more efficient than implementation A. Why?

Chapter 8.4

Priority Queues

Interface/Behaviour Specification

...of the ADT priority queue, named PQueue (user-visible):

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

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

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

Note: Each entry of a priority queue has a priority associated with it. The dequeue operation always removes the entry with the highest (or lowest) priority, which is ensured by the enqueue operation, which places a new element according to its priority in a queue.

Implementation

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

```
newtype PQueue a    = PQ [a]

emptyPQ             = PQ []

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

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

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

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

Verification

Specifier and implementer of the ADT priority queue need to show, respectively:

- The laws of the behaviour specification of the ADT priority queues are consistent and complete.
- The implementation satisfies the laws of the behaviour specification of the ADT priority queue.

...where the specification of the laws was left as homework.

Chapter 8.5

Tables

Chapter 8.5.1

Tables as Functions and Lists

Interface/Behaviour Specification

…of the ADT table, named Table (user-visible):

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

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

-- Behaviour Spec.: Laws for table operations
   Intuitively:
    -- new_T assoc_list: create a new table and ini-
    --   tialize it with the data of assoc_list.
    -- find_T tab ind: retrieve information stored in
    --   table tab at index ind.
    -- upd_T (ind,val) tab: update the entry of table
    --   tab stored at index ind with value val.
   Details: Homework!
```

Implementation A

...of the ADT table as a function (user-invisible):

```
newtype Table a b = Tbl (b -> a)

new_T assoc_list =
 foldr upd_T
       (Tbl (_ -> eror "Item not found"))
       assoc_list

find_T (Tbl f) index = f index

upd_T (index,value) (Tbl f) = Tbl g
 where g j | j==index  = value
           | otherwise = f j
```

Implementation B

…of the ADT table as a new type (user-invisible):

```
newtype Table a b = Tbl [(b,a)]

new_T assoc_list = Tbl assoc_list

find_T (Tbl []) i = error "Item not found"
find_T (Tbl ((j,value):r)) index
  | index==j  = value
  | otherwise = find_T (Tbl r) index

upd_T e (Tbl []) = Tbl [e]
upd_T e'@(index,_) (Tbl (e@(j,_):r))
  | index==j    = Tbl (e':r)
  | otherwise   = Tbl (e:r')
  where Tbl r' = upd_T e' (Tbl r)
```

Verification

Specifier and implementer of the ADT table need to show, respectively:

- The laws of the behaviour specification of the ADT table are consistent and complete.
- The implementation satisfies the laws of the behaviour specification of the ADT table.

...where the specification of the laws was left as homework.

Chapter 8.5.2

Tables as Arrays

Interface/Behaviour Specification

...of the ADT table, named Table′ (user-visible):

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

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

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

Note: The signatures of the table operations have been enlarged by the context `(Ix b) =>` in order to be prepared for array manipulations.

Implementation

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

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

new_T' assoc_list = Tbl' (array (low,high) assoc_list)
 where indices    = map fst assoc_list
       low        = minimum indices
       high       = maximum indices

find_T' (Tbl' a) index          = a ! index

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

Note

- new_T′ takes an association list of index/value pairs and returns the corresponding table.

  To this end, new_T′ determines first the list of indices indices of association list assoc_list, and based on this the boundaries of the new table array by computing the minimum low and the maximum high index of assoc_list; afterwards it constructs the new table array applying the function array to the pair of array bounds (low,high) and association list assoc_list.
- find_T′ and upd_T′ are used to retrieve and update values in the table array, respectively. Note that find_T′ returns a system error, not a user error, when applied to an invalid index.

Verification

Specifier and implementer of the ADT table need to show, respectively:

- The laws for table are consistent and complete.
- The implementation satisfies the laws of the ADT operations of the ADT table.

...whose specification was left as homework.

Chapter 8.6

Displaying ADT Values in Haskell

Displaying ADT Values

...is often necessary but requires some special care, especially in Haskell.

The reasons for this are twofold:

1. ADT values can only be accessed using the ADT operations. Usually, it is crude and cumbersome to display all values of a complex ADT value like a stack or a queue using only the ADT operations, e.g., by completely popping a whole stack.
2. Displaying ADT values straightforwardly in terms of their CDT representations can reveal the internal structure of the CDT breaking the ADT principles of information hiding and (possibly) security.

In Haskell

...breaking the principles of information hiding and (possibly) security always happens if the CDT implementing an ADT is made an instance of the type class Show using an automatic

- `deriving`-clause

which is demonstrated next considering stacks for illustration.

The Problem: Automatic deriving-Clauses

...are unsafe:

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

newtype Stack a = Stk [a] deriving Show

type Stack a    = [a] -- Lists are instance of Show;
                      -- hence, no deriving clause
                      -- required.
```

because displaying stack values reveals their internal structure:

```
push 3 (push 2 (push 1 emptyS))
  ->> Stk 3 (Stk 2 (Stk 1 Empty))

push 3 (push 2 (push 1 emptyS))
  ->> Stk [3,2,1]

push 3 (push 2 (push 1 emptyS))
  ->> [3,2,1] ->> (3:2:1:[])
```

A Note on Information Hiding and Security (1)

Information hiding

- is broken for all three implementation variants as algebraic type, new type, and type alias: Displaying stack values discloses their internal structure and data constructors.

Security

- is broken for the variant as type alias: All list operations are immediately available to create, access, and manipulate stack values using arbitrary list operations. Therefore, type aliases of basic types are not considered valid ADT implementations.
- is preserved for the variants as algebraic type and new type: This is because the data value constructors `Empty` and `Stk` are not exported from the module. A user of the module can thus not use or create a stack value by any other way than the operations exported by the module.

A Note on Information Hiding and Security (2)

This holds anlogously for the other ADT implementations:

Stacks

```
data Stack a    = Empty
                | Stk a (Stack a) deriving Show
newtype Stack a = Stk [a] deriving Show
type Stack a    = [a]
```

Queues and Priority Queues

```
newtype Queue a  = Q [a] deriving Show
newtype PQueue a = PQ [a] deriving Show
```

Tables

```
newtype Table a b = Tbl [(b,a)] deriving Show
newtype Table a b = Tbl (Array b a) deriving Show
```

...straightforward and easy but unsafe and (possibly) insecure.

Remedy: Explicit instance-Declarations (1)

...the safe, secure, and thus recommended way for displaying ADT values, here stacks:

A)
```
instance (Show a) => Show (Stack a) where
  showsPrec _ Empty str = showChar ‘-‘ str
  showsPrec _ (Stk x s) str
    = shows x (showChar ‘|‘ (shows s str))
```

B)
```
instance (Show a) => Show (Stack a) where
  showsPrec _ (Stk []) str = showChar ‘-‘ str
  showsPrec _ (Stk (x:xs)) str
    = shows x (showChar ‘|‘ (shows (Stk xs) str))
```

C)
```
instance (Show a) => Show (Stack a) where
  showsPrec _ [] str = showChar ‘-‘ str
  showsPrec _ (x:xs) str
    = shows x (showChar ‘|‘ (shows xs str))
```

Remedy: Explicit `instance`-Declarations (2)

This way, the very same output for all 3 implementations:

```
push 3 (push 2 (push 1 emptyS)) ->> 3|2|1|-
```

No implementation details about the internal data structure are disclosed:

- Independently of the chosen implementation A, B, (or C), the output is the same.
- Hence, the actually chosen implementation of the ADT `Stack` remains hidden. It is not disclosed to the user (of the module).

Note: The first argument of `showsPrec` is an unused precedence value.

Challenge: Displaying Tables

...represented as functions because there is no general meaningful way to display a function. An instance declaration for

```
newtype Table a b = Tbl (b -> a)
```

for the type class `Show` could thus be chosen minimal/trivial:

```
instance Show (Table a b) where
 showsPrec _ _ str = showString "<<A Table>>" str
```

Chapter 8.7

Summary

Abstract Data Types

...are not a first-class citizen in Haskell.

Nonetheless, specifying and implementing ADTs using modules ensures all three design goals strived for with ADTs:

- Separation of concerns: Separation of specification (interface and behaviour specification) and implementation of a data type (in terms of a CDT and CDT operations matching the ADT operations).
- Information hiding: No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.
- Security: CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated by using the ADT operations implemented by their CDT counterparts.

Note

Due to limitations of the module concept in Haskell, the

- behaviour specification of ADTs can only be given as comments.

If ADT values need to be displayed, this can be done by

- by making the underlying CDT a member of the type class `Show`.

This should always be done by means of an explicit

- `instance`-declaration

since a (more convenient) `deriving`-clause, if possible, would reveal the internal representation of the CDT values, especially the data constructors of the CDT breaking the information hiding principle of ADTs (though the constructors could not be used by a user since they are not exported from the module).

Benefits of Using Abstract Data Types

...evolve directly from the 'by-design built-in' ADT properties:

- Separation of concerns, i.e., the separation of the specification and implementation of a data type

enables

- Information hiding: Only the interface and the behaviour specification of the ADT are publicly known; its implementation as a CDT and operations on it are hidden.

This ensures:

- Security of the data (structure) and its data values from uncontrolled, unintended, or not permitted access.

Altogether, this enables:

- Simple exchangeability of the CDT implementation of an ADT (e.g., simplicity vs. scalability/performance).
- Modularization and programming-load sharing supporting programming-in-the-large.

Relevance of Abstract Data Types

...there are many more examples of data structures, which can be specified and implemented in terms of abstract data types in order to benefit from the built-in ADT properties such as separation of concerns, information hiding, security, exchangeability, modularity, etc., including

- Sets
- Heaps
- Trees (binary search trees, balanced trees,...)
- ...

and also

- Arrays

as illustrated next.

Arrays as Abstract Data Type in Haskell (1)

```
module Array (
        module Ix, -- export all of Ix (for convenience)
        Array, array, listarray (!), bounds, indices,
        elems, assocs, accumArray, (//),
        accum, ixmap ) where

import Ix
infixl 9 !, // ... -- Operator precedence
data (Ix a) => Array a b = ... -- Abstract

array     :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!)       :: (Ix a) => Array a b -> a -> b
bounds    :: (Ix a) => Array a b (a,a)
indices   :: (Ix a) => Array a b -> [a]
elems     :: (Ix a) => Array a b -> [b]
assocs    :: (Ix a) => Array a b -> [(a,b)]
```

Arrays as Abstract Data Type in Haskell (2)

```
accumArray :: (Ix a) => (b -> c -> b) -> b
                        -> (a,a) -> [(a,c)] -> Array a b
(//)       :: (Ix a) => Array a b -> [(a,b)]
                                  -> Array a b
accum      :: (Ix a) => (b -> c -> b) -> Array a b
                              -> [(a,c)] -> Array a b
ixmap      :: (Ix a, Ix b) => (a,a) -> (a -> b)
                             -> Array b c -> Array a c

instance Functor (Array a) where...
instance (Ix a, Eq b)  => Eq (Array a b) where...
instance (Ix a, Ord b) => Ord (Array a b) where...
instance (Ix a, Show a, Show b)
                => Show (Array a b) where...
instance (Ix a, Read a, Read b)
                => Read (Array a b) where...
```

Arrays as Abstract Data Type in Haskell (3)

For the definition of the functions and instance declarations of the module `Array`, see:

– Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 173-178, 2003. (Chapter 16, Arrays)

Chapter 8.8

References, Further Reading

Chapter 8: Basic Reading (1)

Origins

- 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.
- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

Basics, Fundamentals

- Manoochehr Azmoodeh. *Abstract Data Types and Algorithms*. Macmillan Education, 1988.

Chapter 8: Basic Reading (2)

Textbook Representations using Haskell

- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2nd edition, 1998. (Chapter 8, Abstract data types)
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.5, Abstract Types and Modules)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract Data Types)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 16, Abstract data types)

Chapter 8: Selected Further Reading

Handbook Representations, Beyond Haskell

- 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)

- 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)

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 emphasizes the essence of the underlying algorithmic principle.

Pragmatically

- this makes these algorithmic principles easily re-usable.

Outline

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

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

Chapter 3.1

Divide-and-Conquer

Divide and Conquer

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea:

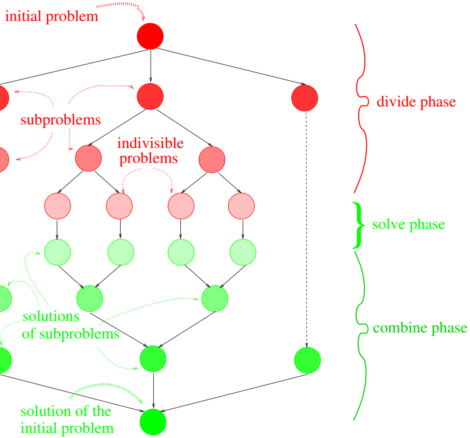
- If a problem instance is simple/small enough, solve it: directly or by means of some basic algorithm.
- Otherwise, divide the problem instance into smaller subproblem instances by applying the division strategy recursively until all subproblem instances are simple enough to be solved directly.
- Combine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

Applicability Requirement:

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

Illustrating the Divide-and-Conquer Principle

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



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

Implementing Divide-and-Conquer as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Divide-and-Conquer as HoF (2)

The arguments of `divide_and_conquer`:

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

Implementing Divide-and-Conquer as HoF (3)

The HoF Implementation:

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

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

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

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

Typical Applications of Divide-and-Conquer

Application fields such as

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

Especially

- Quicksort
- Mergesort
- Binomial coefficients
- ...

Example: Quicksort

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

Counterexample: Fibonacci Numbers (Pitfall!)

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

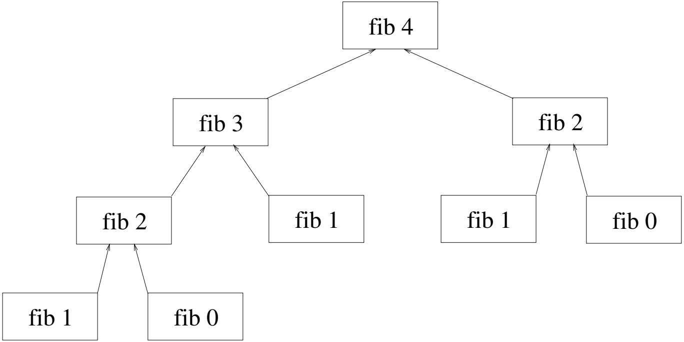
Consider:

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

...shows exponential runtime behaviour due to recomputations!

Illustrating

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



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

Chapter 3.2

Backtracking Search

Backtracking Search

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea:

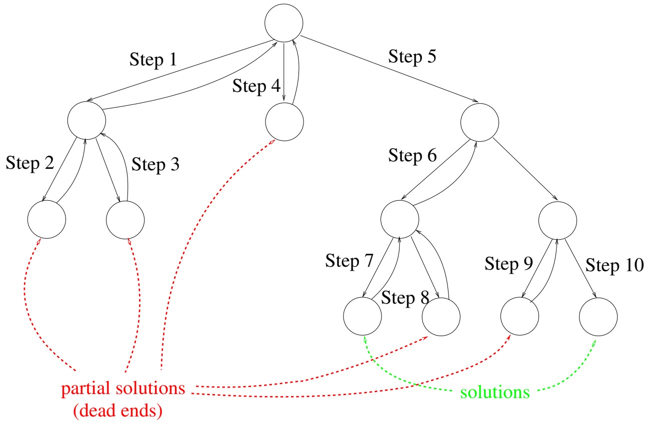
– Search for a particular solution of the problem by a systematic trial-and-error exploration of the solution space.

Applicability Requirements:

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

Illustrating the Backtracking Search Principle

...general stages of a backtracking algorithm:



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

Illustrating Backtracking Search (Cont'd)

Underlying assumptions

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

Note

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

Implementing Backtracking Search as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Backtracking Search as HoF (2)

Note

- Often, the search space is large.

In such cases, the graph forming the search space

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

This requires

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

Implementing Backtracking Search as HoF (3)

Implementation assumptions:

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

The arguments of `search_dfs`:

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

Implementing Backtracking Search as HoF (4)

The HoF Implementation:

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

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

Interface and Behaviour Specification

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

```
module Stack (Stack,empty,is_empty,push,pop,top)
                                                  where

-- Interface Spec.: Signatures of stack operations
empty    :: Stack a
is_empty :: Stack a -> Bool
push     :: a -> Stack a -> Stack a
pop      :: Stack a -> Stack a
top      :: Stack a -> a

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

Typical Applications of Backtracking Search

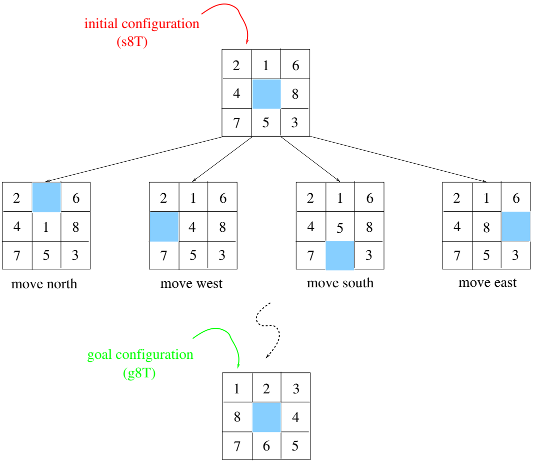
Application fields such as

- Knapsack problems
- Game strategies
- ...

Especially

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

Example: The Eight-Tile Problem (8TP)



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

A Backtracking Search Impl. for 8TP (1)

Modeling the board:

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

The initial board (initial configuration):

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

The final board (goal configuration):

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

A Backtracking Search Impl. for 8TP (2)

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

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

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

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

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

A Backtracking Search Impl. for 8TP (3)

Modeling nodes in the search graph:

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

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

The successor function:

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

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

A Backtracking Search Impl. for 8TP (4)

The goal function:

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

Putting things together:

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

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

Chapter 3.3

Priority-first Search

Priority-first Search (1)

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea

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

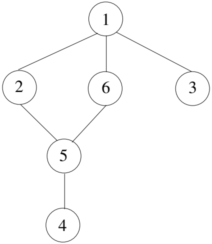
Note: While plain backtracking search proceeds unguidedly and can thus be considered blind, priority-first search/best-first search benefits from (hopefully accurate) information pointing it towards the 'most promising' node.

Priority-first Search (2)

Applicability Requirements

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

Illustrating Different Search Strategies



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

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

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

Implementing Priority-first Search as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Priority-first Search as HoF (2)

Implementation assumptions:

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

The arguments of `search_pfs`:

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

Implementing Priority-first Search as HoF (3)

The HoF Implementation:

$$\texttt{search\_pfs :: (Ord node) => } \underbrace{\texttt{(node -> [node])}}_{\textit{Computing successors}} \texttt{ ->}$$

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

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

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

Interface and Behaviour Specification

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

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

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

-- Behaviour Spec.: Laws for priority queue operations
             -- cf. Chapter 8.4 for different
             -- implementations of priority queues.
```

Typical Applications of Priority-first Search

Application fields such as

- Game strategies
- ...

Especially

- The eight-tile problem
- ...

Example: A Priority-first Search for 8TP

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

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

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

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

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

Chapter 3.4

Greedy Search

Greedy Search (1)

Given: A problem instance P.

Sought: A solution S of P.

Algorithmic Idea

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

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

Greedy Search (2)

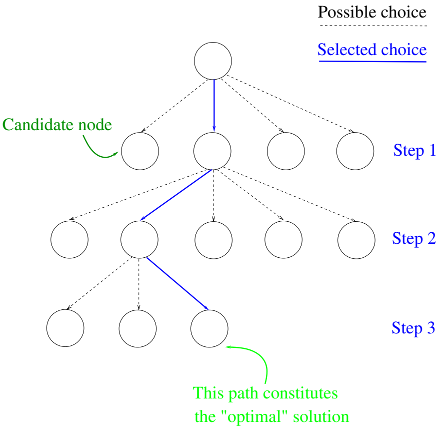
Applicability Requirements

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

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

Illustrating the Greedy Search Principle

...successive stages of a greedy algorithm:



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

Implementing Greedy Search as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Greedy Search as HoF (2)

Implementation assumptions:

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

The arguments of `search_greedy`:

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

Implementing Greedy Search as HoF (3)

The HoF Implementation:

$$\texttt{search\_greedy :: (Ord node) => }\underbrace{\texttt{(node -> [node])}}_{\textit{Computing successors}}\texttt{ ->}$$

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

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

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

Note

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

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

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

Typical Applications of Greedy Search

Application fields such as

– Graph algorithms

– ...

Especially

– Prim's minimum spanning tree algorithm

– The money change problem (MCP)

– ...

Example: A Greedy Search for MCP (1)

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

Modeling coins:

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

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

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

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

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

Example: A Greedy Search for MCP (2)

The goal function:

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

Putting things together:

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

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

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

Chapter 3.5

Dynamic Programming

Dynamic Programming

Given: A problem instance P.

Sought: A solution S of P.

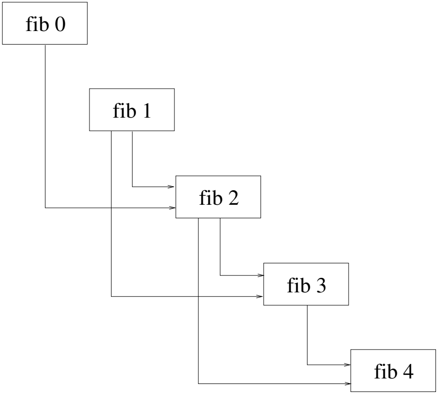
Algorithmic Idea

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

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

Illustrating Dynamic Programming for `fib`

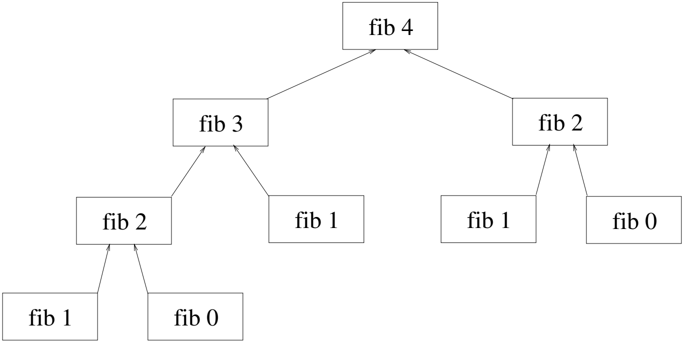
...the dynamic programming computation of the Fibonacci numbers (no recomputation of solutions of subproblems!):



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

Illustrating Divide-and-Conquer for `fib`

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



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

Implementing Dynamic Programming as HoF (1)

Setting:

A problem with

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

Objective:

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

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

Implementing Dynamic Programming as HoF (2)

The arguments of `dynamic`:

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

Implementing Dynamic Programming as HoF (3)

The HoF Implementation:

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

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

Interface/Behaviour Specification

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

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

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

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

Implementation

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

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

Note:

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

Typical Applications of Dynamic Programming

Application fields such as

- Graph algorithms
- Search algorithms
- ...

Especially

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

Example: Computing Fibonacci Numbers

Defining the problem-dependent parameters:

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

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

Putting things together:

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

Chapter 3.6

Dynamic Programming vs. Memoization

Dynamic Programming vs. Memoization

Overall

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

Conceptual difference

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

Minor Benefits of Dynamic Programming

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

Minor Benefits of Memoization

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

Chapter 3.7

References, Further Reading

Chapter 3.1–3.4: Basic Reading

Divide and Conquer, Greedy Algorithms, Memoziation in Haskell

- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 6.4, Divide and Conquer; Chapter 6.5, Search and Enumeration)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract data types; Chapter 8, Top-down design techniques)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 19.6, Avoiding recomputation: memoization – Greedy algorithms)

Chapter 3.1–3.4: Selected Further Reading (1)

Divide and Conquer beyond Haskell

- 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)
- Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 5, Divide and Conquer)
- Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002. (Kapitel 5, Rekursion - Teile und Herrsche)
- Steven S. Skiena. *The Algorithm Design Manual*. Springer-V, 1998. (Chapter 3.6, Divide and Conquer)

Chapter 3.1–3.4: Selected Further Reading (2)

Backtracking beyond Haskell

- James R. Bitner, Edward M. Reingold. *Backtrack Programming Techniques*. Communications of the ACM 18(11):651-656, 1975.
- Gunter Saake, Kai-Uwe Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 4. überarbeitete Auflage, 2010. (Kapitel 8.4, Rekursion: Backtracking)
- Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002. (Kapitel 44, Erschöpfendes Durchsuchen - Backtracking)

Chapter 3.1–3.4: Selected Further Reading (3)

Greedy Algorithms beyond Haskell

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. (Chapter 16, Greedy Algorithms)
- Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 4, Greedy Algorithms; Chapter 5, Divide and Conquer)
- Gunter Saake, Kai-Uwe Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 4. überarbeitete Auflage, 2010. (Kapitel 8.2, Algorithmenmuster: Greedy)

Chapter 3.5–3.6: Basic Reading

Dynamic Programming, Memoization in Haskell

- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract data types; Chapter 9, 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 3.5–3.6: Selected Further Reading (1)

Dynamic Programming, Memoization beyond Haskell

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

Chapter 3.5–3.6: Selected 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 3.5–3.6: Selected Further Reading (3)

- 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)
- Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998. (Chapter 3.1, Dynamic Programming; Chapter 3.2, Limitations of Dynamic Programming)

Final Note

...for additional information and details refer to

- full course notes

available at the homepage of the course at:

http://www.complang.tuwien.ac.at/knoop/
ffp185A05_ss2020.html