Script:Searching and sorting in Haskell

Headline

Lecture "Searching and sorting in Haskell" as part of Course:Lambdas in Koblenz

Summary

We show how to approach the basic algorithmic problems of search and sorting in Language:Haskell. In this manner, we establish proper familiarity with basic functional programming on lists in Haskell, with the use of recursion, local scope, and polymorphism. We also encounter divide and conquer algorithms in this way. Along the way, we discuss some bits of expressiveness of the Haskell type system including support for polymorphism with type constraints, type checking, and type inference.

The discussed encodings of search and sorting do not intend to be the most efficient ones (in Haskell); instead, the intention is to demonstrate basic algorithmic problem solving in Haskell and to provide evidence for Haskell's fitness for describing algorithms declaratively and concisely.

Material

Simple algorithms in Haskell

Concepts

Recap

- Algorithm
- Algorithmic problem
- Search problem
- Search algorithm
- Linear search
- Local scope

Additions

- Binary search
- Sorting problem
- Sorting algorithm
- Insertion sort
- Divide and conquer algorithm
- Quicksort
- Selection sort
- Merge sort
- Median
- Type system
- Type signature
- Polymorphism
- Type checking
- Type inference

Languages

- Language:Haskell

Features

- Feature:Median

Contributions

- Contribution:haskellBarchart

Metadata

- Course:Lambdas in Koblenz
- Script:Basic software engineering for Haskell

Binary search

Headline

Solve the search problem for sorted input

Description

Semi-formally, binary search can be described by an algorithm as follows:

- Given is a sorted list *l* and a value *v* of the element type of *l*.
- If *l* is the empty list then return *False*.
- Let *m* be the element in the "middle" of *l*.
- If *m* equals *v*, then return *True*.
- If *m* < *v*, then recursively search *v* in the list containing all elements to the right of *m*.
- If *m* > *v*, then recursively search *v* in the list containing all elements to the left of *m*.

Please note that this formulation also expresses that the element type must admit comparison for comparison.

Illustration

Binary search

```
-- Polymorphic binary search
-- Assume that input list is sorted
search :: Ord a => [a] -> a -> Bool
search [] _ = False
search xs x =
   if x < y then search ys1 x
   else if x > y then search ys2 x
   else True
  where
   ys1 = take l xs
   (y:ys2) = drop l xs
   l = length xs `div` 2
```

The implemented search function can be applied as follows:

```
-- Illustrate binary search
main = do
  let input = [1,2,3,4,4]
  print $ search input 1 -- True
  print $ search input 5 -- False
```

Note that the input list is readily sorted. The less efficient, linear search does not require the input list to be sorted.

We may also represent the input as a sorted tree. In this manner, the decision of going to the left or the right side after "halving" is much more straightforward. Here is a suitable algebraic data type for trees:

```
-- Binary trees
data Tree a = Empty | Fork a (Tree a) (Tree a)
```

The corresponding search function looks like this:

```
-- Polymorphic binary search
-- Assume that input tree is sorted
search :: Ord a => Tree a -> a -> Bool
search Empty _ = False
search (Fork x1 l r) x2 =
  if x2 < x1
    then search l x2
    else if x2 > x1
      then search r x2
      else True
```

A demo follows:

```
-- Illustrate binary search
main = do
```

```
let input = Fork 3 (Fork 1 Empty Empty) (Fork 42 Empty Empty)
print $ search input 1 -- True
print $ search input 3 -- True
print $ search input 42 -- True
print $ search input 88 -- False
```

Citation

(http://en.wikipedia.org/wiki/Binary_search_algorithm, 21 April 2013)

In computer science, a binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array. ... In each step, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

Metadata

- http://en.wikipedia.org/wiki/Binary_search_algorithm
- Search algorithm
- Search problem
- Linear search

Algorithm

Headline

A step-by-step procedure for achieving a certain IO behavior

Illustration

See greatest common divisor and linear search for examples of an algorithm which is described both semi-formally and in an actual programming language.

Citation

(http://en.wikipedia.org/wiki/Algorithm, 14 April 2013)

In mathematics and computer science, an algorithm ... is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

More precisely, an algorithm is an effective method expressed as a finite list... of well-defined instructions... for calculating a function. ... Starting from an initial state and initial input (perhaps empty),... the instructions describe a computation that, when executed, proceeds through a finite ... number of well-defined successive states, eventually producing "output"... and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Metadata

- http://en.wikipedia.org/wiki/Algorithm
- Vocabulary:Programming
- Program
- Algorithmic problem
- Namespace:Concept

Algorithmic problem

Headline

A problem that can be solved with an algorithm

Illustration

- Examples of algorithmic problems: greatest common divisor, the factorial, and the search problem.
- An example of non-algorithmic problems: the Halting problem.

Metadata

- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Algorithm
- Concept

Median

Headline

The value in the "middle" of a sorted sequence

Citation

(http://en.wikipedia.org/wiki/Median, 21 April 2013)

In statistics and probability theory, the median is the numerical value separating the higher half of a data sample, a population, or a probability distribution, from the lower half. The median of a finite list of numbers can be found by arranging all the observations from lowest value to highest value and picking the middle one (eg, the median of {3, 5, 9} is 5). If there is an even number of observations, then there is no single middle value; the median is then usually defined to be the mean of the two middle values,... which corresponds to interpreting the median as the fully trimmed mid-range.

Metadata

- http://en.wikipedia.org/wiki/Median
- Vocabulary:Mathematics
- Concept

Merge sort

Headline

The Merge sort sorting algorithm

Citation

(http://en.wikipedia.org/wiki/Merge_sort, 21 April 2013)

Conceptually, a merge sort works as follows

1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

Illustration

See the visualization of Merge sort on Wikipedia:

http://en.wikipedia.org/wiki/Merge_sort

See various illustrations of Merge sort as available on YouTube, e.g.:

Merge-sort with Transylvanian-saxon (German) folk dance

Recursive merge sort in Java

```
public static void mergeSort(int[] a) {
 int[] temp = new int[a.length];
 mergeSort(a, temp, 0, a.length - 1);
}

public static void mergeSort(int[] a, int[] temp, int min, int max) {
 // Base case
 if (!(min < max)) return;

 // Split array and solve sub-problems recursively
 int middle = (min + max) / 2;
 mergeSort(a, temp, min, middle);
 mergeSort(a, temp, middle + 1, max);

 // Merge via temporary array
 merge(a, temp, min, middle, max);
}

public static void merge(int[] a, int[] temp, int min, int middle, int max) {
 int i = min; // loop over left half
 int j = middle + 1; // loop over right half
 int k = min; // loop over merged result
 while (k <= max)
  temp[k++] = (i <= middle && (j > max || a[i] < a[j])) ?
      a[i++] // copy from left half
    : a[j++]; // copy from right half
 // Override array by merged tempory result
 for (k = min; k <= max; k++)
  a[k] = temp[k];
}
```

Recursive merge sort in Haskell

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort [] = []
sort [x] = [x]
sort xs = merge (sort ys) (sort zs)
  where
   (ys,zs) = split xs

-- Split a list into halves
```

```
split :: [a] -> ([a],[a])
split xs = (take len xs, drop len xs)
  where
    len = length xs `div` 2

-- Merge sorted sublists
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
  if x<=y
    then x : merge xs (y:ys)
    else y : merge (x:xs) ys
```

The main sorting function relies on helpers for splitting the input lists into halves and for merging sorted sublists. An empty list as much as a singleton lists are immediately sorted, as modeled by the first two equations of *sort*. The *split* helper uses list-processing goodies *take* and *drop* to extract two halves (+/- one element for a list of odd length). The*merge* help offers two base cases for a merge to trivially complete, if either of the two operands is an empty list; the recursive case compares the heads of both operands to decide on which of them goes first into the merged result.

The implementation is exercised as follows:

```
main = do
  let input = [2,4,3,1,4]
  print $ sort input -- [1,2,3,4,4]
```

Metadata

- http://en.wikipedia.org/wiki/Merge_sort
- Sorting algorithm
- Divide and conquer algorithm

Polymorphism

Headline

The ability of program fragments to operate on elements of several types

Illustration

Consider the type of list append inLanguage:Haskell:

```
(++) :: [a] -> [a] -> [a]
```

This type signature uses a type variable *a* to express that list append is polymorphic in the element type*a*. The operation can be applied for as long as the element type of both operand lists for an append are the same. We also speak of parametric polymorphism in this case.

Consider the type of addition inLanguage:Haskell:

```
(+) :: Num a => a -> a -> a
```

This type signature uses a type constraint on the operand type of addition to express that only "suitable" types (i.e., type-class instances of *Num*) can be used for addition. We also speak oftype-class polymorphism or more generally ofbounded polymorphism in this case. Languages withsubtyping may also use types in a subtyping hierarchy for bounds.

Metadata

- http://en.wikipedia.org/wiki/Polymorphism (computer science)
- http://en.wikipedia.org/wiki/Polymorphism in object-oriented programming
- Vocabulary:Programming language
- Concept

Recursion

Headline

The use of self-reference in defining abstractions

Illustration

Clearly, there are different forms of recursions, as they are different abstraction mechanisms that permit recursion. For instance, in functional programming, both Functions and data types may be defined recursively.

Recursive functions

Consider the following recursive formulation of the factorial function in Language:Haskell:

```
-- A recursive definition of the factorial function
factorial n =
  if n==0
    then 1
    else n * factorial (n-1)
```

This is essentially a form of primitive recursion: the function definition checks for the argument *n* to be "0" for the base case and applies the function recursively to the predecessor of the argument otherwise. For the record, non-recursive formulations are feasible, too, depending on the helper functions we are willing to use. For instance, we may use the ".." operator to enumerate all values in a range and then apply the *product* function:

```
-- A non-recursive definition of the factorial function
factorial' n = product [1..n]
```

Recursive data types

Under construction.

Metadata

- Vocabulary:Programming theory
- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Recursion_(computer_science)
- http://mathworld.wolfram.com/Recursion.html
- Concept

Search algorithm

Headline

An algorithm to solve the search problem

Illustration

Consider the following list of ints as the input for searching:

```
[2,4,3,1,4]
```

A search algorithm returns *True* if asked to search for 1 and *False* for 5.

See linear search and binary search for specific search algorithms.

Citation

(http://en.wikipedia.org/wiki/Search_algorithm, 14 April 2012)

In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph.

Metadata

- http://en.wikipedia.org/wiki/Search_algorithm
- Algorithm
- Search problem

Sorting problem

Headline

The problem of sorting a given list

Description

The problem can be described, for example, as follows:

- Input:
  - A list *l* of values
- Output:
  - A list *r* satisfying these properties:
    - *r* is a permutation is of *l*.
    - *r* is sorted.

For instance, given the input {2,4,3,1,4}, the output should be {1,2,3,4,4}.

This is an algorithmic problem; there are various sorting algorithms, e.g., Quicksort.

Illustration

Sorting problem in Haskell

The ingredients of the above problem description, i.e., the test for two lists to be permutations of each other and the test for a list to be sorted, can be described more formally as follows; we use Language:Haskell notation:

```
-- Test for two lists to be permutations of each other
permutation :: Eq a => [a] -> [a] -> Bool
permutation [] [] = True
permutation (x:xs) ys = remove ys []
  where
    -- Repeat removal of equal elements
    remove [] _ = False
    remove (y:ys) zs =
      if (x==y)
        then permutation xs (zs++ys)
        else remove ys (y:zs)

-- Test for a list to be sort
sorted :: Ord a => [a] -> Bool
sorted [] = True
sorted [x] = True
sorted (x1:x2:xs) = x1 <= x2 && sorted (x2:xs)
```

We apply the list properties to a few sample lists:

```
-- Illustrate list properties
main = do
  let l1 = [2,4,3,1,4]
  let l2 = [1,2,3,4,4]
  let l3 = [1,8,2,7,4]
  print $ sorted l1 -- False
  print $ sorted l2 -- True
  print $ sorted l3 -- False
  print $ permutation l1 l2 -- True
  print $ permutation l1 l3 -- False
```

Metadata

- https://en.wikipedia.org/wiki/Sorting_algorithm
- Algorithmic problem

Type constraint

Headline

A means of constraining a polymorphic type

Illustration

The notion of type constraint is pretty general as it occurs in different ways in programming languages which is why we do not attempt a comprehensive description here. Instead, we briefly compare how type constraints show up in Java versus Haskell.

We use the following problem to illustrate type constraints: Suppose you want to count the elements in an array that are *greater* than a given element. This sort of function (or method) would need to be parametric in the element type of the array and that type would also need to be constrained to admit comparison.

Type comstraints in Java

The type signature for a suitable method for the problem at hand is of the following form:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem)
```

Type comstraints in Java

The type signature for a suitable function for the problem at hand is of the following form:

```
countGreaterThan :: Ord t => [t] -> t -> Ordering
```

Haskell's type constraints are discussed in more detail in the broader context oftype signatures.

Metadata

- Vocabulary:Programming language
- http://msdn.microsoft.com/en-us/library/d5x73970.aspx
- http://en.wikipedia.org/wiki/Generic_programming
- http://docs.oracle.com/javase/tutorial/java/generics/bounded.html
- Concept

Type signature

Headline

The type associated with an identifier

Synonyms

Other terms may be used for referring to type signatures in the context of specific programming paradigms and programming languages, e.g.:

- Function signature
- Type annotation
- Method signature (OO programming)
- Function prototype (Language:CPlusPlus)

Description

Type signatures in functional programming

All expressions and all named functions for that matter have an associated type. When such a type is explicitly assigned (declared), then we speak of a type signature. When the type system of the language at hand permits inference, as in the case of Language:Haskell, for example, then the type signature may also be omitted; one may nevertheless infer the type of the defined function.

Illustration

Type signatures in Haskell

We investigate some type signatures of predefined functions at the Haskell prompt. To this end, we use the interpreter command *:t" to ask for types of expressions. (User input at the Haskell prompt is prefixed by ">"; the result is shown in the subsequent line(s).)*

Here is the type signature of Boolean negation.

```
> :t not
not :: Bool -> Bool
```

The type signature declares that the function *not* takes an argument of type *Bool* and returns a result *Bool*.

Hre is the type signature of conjunction ("&&").

```
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

By enclosing "&&" into parentheses, we deal with the fact that "&&" is an infix operator. The type signature declares that the function "&&" takes two arguments of type *Bool* and returns a result of type *Bool*.

Types and function signatures for that matter may be polymorphic; this can be seen from the occurrence of type variables (starting in lower case). Let's look at some examples at the Haskell prompt.

The totally *undefined* function is of a completely polymorphic type.

```
> :t undefined
undefined :: a
```

The identity function (*id*) is polymorphic with the same type variable for domain and range.

```
> :t id
id :: a -> a
```

The projection function for the first component of a pair (*fst*) is polymorphic in the type of the two components of the pair; the result is of the same type as the first component.

```
> :t fst
```

```
fst :: (a, b) -> a
```

The concatenation function ("++") is polymorphic in the element type of the lists to be concatenated. That is, the function takes two lists of the same polymorphic type and returns a list of the same polymorphic type.

```
> :t (++)
(++) :: [a] -> [a] -> [a]
```

In Haskell, type variables in type signatures may be constrained so that the variables are not universally polymorphic, but they can only be instantiated with types that are instances of appropriate type classes (i.e., sets of types). Without going into detail here, let's look at some examples. Constraints precede argument and result types before an extra arrow "=>".

Equality is not universally polymorphic; it can be applied only to types which define equality; this is modeled by a type class (set of types) *Eq*. For instance, strings can be compared for equality, but functions cannot.

```
> :t (==)
(==) :: Eq a => a -> a -> Bool
> "foo" == "bar"
False
> not == not
... error ... no instance for (Eq (Bool -> Bool)) ...
```

There are several number types (in Haskell). Thus, addition ("+") is overloaded on the grounds of the following signature which refers to a type class *Num*; we also show addition for two different number types: integer (*Int*) and floating point numbers (*Float*).

```
> :t (+)
(+) :: Num a => a -> a -> a
> 20.9+21.1
42.0
> 21+21
42
```

Method signatures in Java

A static method in Language:Java for computing the absolute value of a float may have the following type signature:

```
public static float abs(float a)
```

That is, the method takes an argument of type *float* for the value to be mapped to its absolute value. Further, the method's result type is *float* for the actual absolute value.

An instance method in Language:Java for withdrawing money from an account may have the following type signature:

```
public boolean withdraw(float amount)
```

That is, the method takes an argument of type *float* for the amount to be withdrawn. Further, the method's result type is *boolean* to be able to communicate whether withdrawal was successful or not.

Citation

(http://en.wikipedia.org/wiki/Type_signature, 21 April 2013)

In computer science, a type signature or type annotation defines the inputs and outputs for a function, subroutine or method. A type signature includes at least the function name and the number of its arguments. In some programming languages, it may also specify the function's return type, the types of its arguments, or errors it may pass back.

Metadata

- Vocabulary:Programming
- Vocabulary:Programming language
- http://en.wikipedia.org/wiki/Type_signature
- https://wiki.haskell.org/Type_signature

Type system

Headline

Rules used for type checking or inference

Illustration

Consider a function application in Language:Haskell like this:

```
not True
```

This expression applies logical negation (*not*) to the Boolean literal *True*. (For what it matters, the result of the function application would be *False*.) This expression type-checks because the associated rule of the type system succeeds, which is that **the type of a function application is the result type of the applied function where the actual argument must be of the declared formal argument type**.

Metadata

- Vocabulary:Programming language
- http://en.wikipedia.org/wiki/Type_system
- Concept

Contribution:haskellBarchart

Headline

Analysis of historical company data with Language:Haskell

Characteristics

Historical data is simply represented as list of year-value pairs so that company data is snapshotted for a number of years and any analysis of historical data can simply map over the versions. A simple chart package for Haskell is leveraged to visualize the development of salary total and median over the years. In this manner, the contribution demonstrates how to declare external dependences via Technology:Cabal. Further, the contribution also demonstrates modularization and code organization. In particular, where clauses for local scope and export/import clauses for modularization are used carefully.

Illustration

We would like to generate barcharts as follows:

These barcharts are generated by the following functionality. Given a filename, a title (such as "Total" or "Median") and a year-to-data mapping, generate a PNG file with the barchart for the distribution of the data.

```
-- | Generate .png file for development of median
chart :: String -> String -> [(Int,Float)] -> IO ()
chart filename title values = do
 let fileoptions = FileOptions (640,480) SVG empty
 renderableToFile fileoptions (toRenderable layout) filename
 return ()
   where
    layout
      = def
      & layout_title .~ "Development of salaries over the years"
      & layout_plots .~ [plotBars bars]
    bars
      = def
      & plot_bars_titles  .~ [title]
      & plot_bars_spacing .~ BarsFixGap 42 101
      & plot_bars_style   .~ BarsStacked
      & plot_bars_values  .~ values'
    values'
      = map (\(y,f) -> (y, [float2Double f])) values
```

Metadata

- Feature:Flat company
- Feature:Total
- Feature:Median
- Feature:History
- Contributor:rlaemmel
- http://hackage.haskell.org/package/Chart-0.16
- Contribution:haskellEngineer

Course:Lambdas in Koblenz

Headline

Introduction to functional programming at the University of Koblenz-Landau

Schedule of latest edition

- Lecture First steps
- Lecture Basic software engineering
- Lecture Searching and sorting
- Lecture Basic data modeling
- Lecture Higher-order functions
- Lecture Type-class polymorphism
- Lecture Functors and friends
- Lecture Functional data structures
- Lecture Unparsing and parsing
- Lecture Monads

Additional lectures

- Lecture Generic functions

Metadata

- http://softlang.wikidot.com/course:fp

Language:Haskell

Headline

The functional programming language Haskell

Details

There are plenty of Haskell-based contributions to the 101project. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity.

Metadata

- http://www.haskell.org/
- http://en.wikipedia.org/wiki/Haskell_(programming_language)
- Functional programming language

Insertion sort

Headline

The Insertion sort sorting algorithm

Citation

(http://en.wikipedia.org/wiki/Insertion_sort, 21 April 2013)

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Illustration

See the visualization of Insertion sort on Wikipedia:

http://en.wikipedia.org/wiki/Insertion_sort

See various illustrations of Insertion sort as available on YouTube, e.g.:

Insert-sort with Romanian folk dance

Iterative insertion sort in Java

Given a list of length *n*, in the already sorted prefix 0.*i*-1, elements are moved to the right to make space for the next element *x* from the as yet unsorted postfix*i*..*n*-1 to be inserted in the right position.

```
public static void insertionSort(int[] a) {
 for (int i = 1; i < a.length; i++) {
  int x = a[i];
  int j = i;
  while (j > 0 && a[j - 1] > x) {
   a[j] = a[j - 1];
   a[j - 1] = x;
   j--;
  }
 }
}
```

Recursive insertion sort in Haskell

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort xs = inserts xs []

-- Insert given elements in an emerging result
inserts :: Ord a => [a] -> [a] -> [a]
inserts [] r = r
inserts (x:xs) r = inserts xs (insert x r)

-- Insert a given element in a list
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) =
  if x <= y
    then x : y : ys
    else y : insert x ys
```

The main sorting function relies on two helpers to model sorting as repeated insertion. That is, the*inserts* helper successfully inserts all elements into an emerging result list; the *insert* help perform a single insert operation such that the given element is inserted into a readily sorted list in a way such that the result is sorted, too. The first equation of *insert* models that any value can be trivially inserted into an empty list. The second equation of *insert* compares the given value *x* with the head *y* of the sorted list. If *x<=y* then *x* belongs before *y*; otherwise *y* is maintained as the head and insertion continues into the tail of the list by means of recursion.

The implementation is exercised as follows:

```
main = do
  let input = [2,4,3,1,4]
  print $ sort input -- [1,2,3,4,4]
```

Metadata

- http://en.wikipedia.org/wiki/Insertion_sort
- Sorting algorithm

Linear search

Headline

Solve the search problem by iterating over the input list

Description

Consider the search problem, i.e., the problem of determining whether a given value occurs in a given list. This problem is an algorithmic problem, i.e., it can be solved by an algorithm, e.g., by linear search.

Semi-formally, linear search can be described by an algorithm as follows:

- Given is a list *l* and a value *v* of the element type of *l*.
- Perform the following steps to search *v* in *l*:
  - Initialize an index variable *i* to 0 (to refer to the first element of *l*, if any).
  - Repeat the following steps until a result is returned.
    - Return *False*, if *i* equals the length of *l*.
    - Return *True*, if *l* stores *v* at the index *i*.
    - Increment *i*.

Please note that this formulation also expresses that the element type must admit comparison for equality.

Illustration

Linear search in Haskell

```
-- Polymorphic linear search
search :: Eq a => [a] -> a -> Bool
search [] _ = False
search (x:xs) y = x==y || search xs y
```

The type of the *search* function is polymorphic in that admits arbitrary element types for as long as equality ("Eq") is supported for the type.

The implemented search function can be applied as follows:

```
-- Illustrate linear search
main = do
 let input = [2,4,3,1,4]
 print $ search input 1 -- True
 print $ search input 5 -- False
```

Citation

(http://en.wikipedia.org/wiki/Linear_search, 21 April 2013 with Knuth's "The Art of Computer Programming" credited for citation)

In computer science, linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Metadata

- http://en.wikipedia.org/wiki/Linear_search
- Search algorithm
- Search problem
- Binary search

Divide and conquer algorithm

Headline

An algorithm recursively breaking down a problem

Illustration

See Quicksort and Merge sort for illustrations.

Citation

(http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm, 21 April 2013)

In computer science, divide and conquer (D&C) is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Metadata

- http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm
- Algorithm
- Vocabulary:Programming

Quicksort

Headline

The Quicksort sorting algorithm

Citation

(http://en.wikipedia.org/wiki/Quicksort, 21 April 2013)

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

Illustration

See the visualization of Quicksort on Wikipedia:

http://en.wikipedia.org/wiki/Quicksort

See various illustrations of Quicksort as available on YouTube, e.g.:

Quick-sort with Hungarian (KÃ¼kÃ¼llÅ‘menti legÃ©nyes) folk dance

Quicksort in Haskell

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort [] = []
sort (pivot:rest) =
       (sort lesser)
    ++ [pivot]
    ++ (sort greater)
 where
  lesser  = filter (< pivot) rest
  greater = filter (>= pivot) rest
```

The first equation models that an empty list is sorted vacuously. The second equation picks the head of the list as the 'pivot' element, which is used to partition the input. Indeed, all elements 'lesser' than 'pivot' are collected in one helper list and all elements 'greater' (or equal) than 'pivot' are collected in another helper list. Quicksort is then invoked recursively on 'lesser' and 'greater' and the intermediate results are appended with the 'pivot' element in between.

The implementation is exercised as follows:

```
main = do
 let input = [2,4,3,1,4]
 print $ sort input -- [1,2,3,4,4]
```

Metadata

- http://en.wikipedia.org/wiki/Quicksort
- Sorting algorithm
- Divide and conquer algorithm

Selection sort

Headline

The Selection sort sorting algorithm

Citation

(http://en.wikipedia.org/wiki/Selection_sort, 21 April 2013)

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Illustration

See the visualization of Selection sort on Wikipedia:

http://en.wikipedia.org/wiki/Selection_sort

See various illustrations of Selection sort as available on YouTube, e.g.:

Select-sort with Gypsy folk dance

Selection sort in Haskell

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort xs = selects xs

-- Repeat selection of smallest element
selects :: Ord a => [a] -> [a]
selects [] = []
selects xs = x : selects xs'
  where
    x = smallest (head xs) (tail xs)
    xs' = remove x xs

-- Find smallest element
smallest :: Ord a => a -> [a] -> a
smallest x [] = x
smallest x (y:ys) = smallest (min x y) ys

-- Remove a given element
remove :: Eq a => a -> [a] -> [a]
remove _ [] = error "Element not found for removal."
remove x (y:ys) =
  if x==y
    then ys
    else y : remove x ys
```

The main sorting function relies on helpers to model sorting as repeated selection of the smallest element from the unsorted input. That is, the *selects* helper repeats selection combined with removal of the smallest element; the*smallest* helper determines the smallest element in an unsorted list while starting with its head as the first candidate; the *remove* helper removes a given element from a list following the pattern of linear search.

The implementation is exercised as follows:

```
main = do
  let input = [2,4,3,1,4]
  print $ sort input -- [1,2,3,4,4]
```

Metadata

- http://en.wikipedia.org/wiki/Selection_sort

- Sorting algorithm

Search problem

Headline

The problem of determining whether a given value occurs in a given list

Description

The search problem is an algorithmic problem as follows:

- Input:
  - A list *l* of values
  - A value *v*
- Output:
  - *True*: *v* occurs in *l*.
  - *False*: otherwise.

This is an algorithmic problem because one can obviously define an algorithm to iterate over the elements of *l* and to check for an occurrence of *v* along the way. This simple idea is the foundation for linear search. When additional constraints are imposed on the input, then search may also be more efficient; see binary search for example.

Illustration

See linear and binary search.

Metadata

- http://en.wikipedia.org/wiki/Search_problem
- Algorithmic problem

Sorting algorithm

Headline

An algorithm to solve the sorting problem

Citation

(http://en.wikipedia.org/wiki/Sorting_algorithm, 14 April 2012)

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. ... More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation (reordering) of the input.

Metadata

- http://en.wikipedia.org/wiki/Sorting_algorithm
- Sorting problem
- Concept

Type checking

Headline

Verification of programs or parts thereof to be of the right types

Illustration

A programming language implementation may perform type checking at compile time or prior to interpretation or during run time. Consider, for example, the following function for incrementing ints in Haskell which favors compile-time type checking (or type checking prior to interpretation):

```
inc :: Int -> Int
inc x = x + 1
```

This function type-checks because the function definition is in compliance with the stated type signature. Now assume that the function signature would have been instead this one:

```
inc :: Int -> Bool
inc x = x + 1
```

Obviously, type checking would be bound to fail on this example, as the result type of the function is *Int* rather than *Bool*.

Metadata

- Vocabulary:Programming language
- http://en.wikipedia.org/wiki/Type_system
- http://c2.com/cgi/wiki?TypeChecking
- Concept

Type inference

Headline

The automatic deduction of the type of a program fragment

Illustration

A programming language implementation may perform type inference to compensate for the lack of explicitly declared type signatures such that type checking is still feasible. Typically, type inference happens at compile time. Consider, for example, the following function for negating Booleans in Language:Haskell:

```
not True = False
not False = True
```

The Language:Haskell type system is perfectly confident to infer, even without any type signature at hand, that *not* is of the following type:

```
not :: Bool -> Bool
```

Type inference becomes enormously more powerful, once higher-order functions are taken into account.

Metadata

- Vocabulary:Programming language
- http://en.wikipedia.org/wiki/Type_inference
- Concept

Feature:Median

Headline

Compute the median of the salaries of all employees

Description

Management would like to know the median of all salaries in a company. This value may be used for decision making during performance interviews, e.g., in the sense that any employee who has shown exceptional performance gets a raise, if the individual salary is well below the current median. Further, the median may also be communicated to employees so that they can understand their individual salary on the salary scale of the company. In practice, medians of more refined groups of employees would be considered, e.g., employees with a certain job role, seniority level, or gender.

Motivation

This feature triggers a very basic statistical computation, i.e., the computation of the median of a list of sorted values. Of course, the median is typically available as a primitive or from a library, but when coded explicitly, it is an exercise in list processing. This feature may also call for reuse such that code is shared with the implementation of Feature:Total because both features operate on the list of all salaries.

Illustration

The following code stems from Contribution:haskellStarter:

```
-- Median of all salaries in a company
median :: Company -> Salary
median = medianSorted . sort . salaries
```

First, the salaries are to be extracted from the company. Second, the extracted salaries are to be sorted, where a library function *sort* is used here. Third, the sorted list of salaries is to be processed to find the median.

```
-- Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = getSalaries es

-- Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Salary]
getSalaries [] = []
getSalaries (e:es) = getSalary e : getSalaries es

-- Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary (_, _, s) = s

-- Median of a sorted list
medianSorted [] = error "Cannot compute median on empty list."
medianSorted [x] = x
medianSorted [x,y] = (x+y)/2
medianSorted l = medianSorted (init (tail l))
```

Relationships

- See Feature:Total for another query scenario which also processes the salaries of all employees in a company.

Metadata

- Functional requirement
- Optional feature
- Query