DTU Compute

Lyngby 17-09-2020
Michael R. Hansen
version 2

Exercise: Sorting and property-based testing

You shall develop a simple version of *merge sort*, an interesting sorting algorithm that has an $n \log n$ worst-case execution time. The purpose of this particular exercise is to get practice in the development of elegant functional programs on lists – not to develop efficient sorting programs. Furthermore, you should achieve a basic understanding of computations of recursive functions on lists.

Strive for succinctness and elegance when you solve this problem — it is important that your programs and program designs can be communicated to other people.

Remarks: We shall later in the course study techniques addressing efficiency. Furthermore, there are efficient sorting functions in the .NET libraries, for example, `List.sort`.

You will also touch upon property-based testing, which is a powerful test tool and method for automatic test of correctness properties.

The *merge sort* algorithm can be expressed by a functional composition using two functions: `merge` and `split`, where `merge` combines two sorted lists into a single sorted list, and `split` extracts to lists of almost the same sizes from a given list.

The `merge` function

A *merge* of two sorted lists is a new sorted list made up from the elements of the arguments. For example `merge`([1;4;9;12],[2;3;4;5;10;13]) = [1;2;3;4;4;5;9;10;12;13].

In your declaration of `merge`(xs, ys) you can assume that xs and ys both are ordered, and your declaration must be such that the result is ordered as well. In order words, `merge` must preserved the *invariant* `ordered`.

Exploit the assumption that xs and ys are both ordered in your declaration. Furthermore, give a brief argument showing that the value of `merge`(xs, ys) is ordered when xs and ys are. (One or two lines for each pattern in the declaration should suffice.)

The `split` function

Declare a function to *split* a list into two lists of (almost) the same lengths. You may declare the function `split` such that

$$\texttt{split}\,[x_0; x_1; x_2; x_3; \ldots; x_{n-1}] \;=\; ([x_0; x_2; \ldots], [x_1; x_3; \ldots])$$

This function was one of last week's exercises.

The `sort` function

The idea behind *top-down* merge sort is a recursive algorithm: take an arbitrary list xs with more than one element and split it into two (almost) equal-length lists: xs_1 and xs_2. Sort xs_1 and xs_2 and merge the results. The empty list and lists with just one element are the base cases.

Declare an F# function for top-down merge sort.

Correctness

The result of `sort` xs should be a list $ys = [y_0; y_1; \ldots; y_{n-1}]$ satisfying

1. it is *ordered*, that is $y_0 \leq y_1 \leq \cdots \leq y_{n-1}$, and
2. it consists of exactly the same elements as xs, respecting the number of occurrences as well, that is, if an element x occurs k times in xs, then x must occur k times in ys as well.

Declare an F# function `ordered: int list -> bool` to test whether a list is ordered.

A function that returns a truth value, such as `ordered` is also called a *predicate*.

One part of the correctness properties of sort can be expressed by the predicate:

```
let orderedSort(xs: int list) = ordered(sort xs)
```

To prove correctness one must establish that $orderedSort(xs)$ holds for every integer list xs; but we will not considered proofs of correctness in this course.

Property-based testing

We shall use the tool FsCheck to test properties of .Net programs, in our case FSharp programs. Properties are expressed as predicates, like `orderedSort`, and the tool provides functions that on randomly generated values can test whether a given predicate hold. For example the following piece of code:

```
#I @"C:\Users\mire\.nuget\packages\fscheck\2.14.0\lib\net452"
#r @"FsCheck.dll"
open FsCheck

let commProp(x,y) = x+y = y+x;;
```

```
let commPropTest = Check.Quick commProp;;
let commPropTestVerbose = Check.Verbose commProp;;
```

will test two times whether addition is a commutative operation on 100 randomly generated pairs of integers. The last test will show the 100 test cases.

The above code assumes that FsCheck is installed and that the file `FsCheck.dll` is in the directory: `C:\Users\mire\.nuget\packages\fscheck\2.14.0\lib\net452`.

1. Have a look at `https://fscheck.github.io/FsCheck/` and install FsCheck on your computer.

2. Adapt the path in the above piece of code so that it fits on the placement of FsCheck on your computer and try out property-based testing.

3. Test the property `orderedSort`.

   Notice that `xs` is required to be an integer list in the declaration of `orderedSort`. It is important for FsCheck that the predicates being tested are NOT polymorphic functions.

Further property-based testing

The main activity when doing property-based testing is to program correctness properties; not to make test cases.

You shall now develop programs to address the second property of a sorting program.

By a *counting* for a given list xs we understand a list of the form $[(x_1, c_1); (x_2, c_2); \ldots; (x_k, c_k)]$, where x_j is an integer (occurring in xs) and c_j is a count (a positive integer) for the number of occurrences of x_j in xs. We require that $x_1 < x_2 < \cdots < x_k$, that is there is a unique counting for every list xs.

For example, the counting for $[3; 2; 6; 3; 2; 1]$ is $[(1, 1); (2, 2); (3, 2); (6, 1)]$.

1. Declare a function `increment`(x, cnt). The value of `increment`(i, cnt) is the counting obtained from cnt by incrementing the count for x by one.

2. Declare a function `toCounting` xs, that makes a counting for a given list xs.

3. Use property-based testing to test the property: `ordered(toCounting` xs`)`.

4. Use property-based testing to test the property: `toCounting` xs = `toCounting(sort` xs`)`.