Assignment 2

Programmering og Problemløsning
Department of Computer Science
University of Copenhagen

Victor Vangkilde Jørgensen - kft410

November 2, 2024

1 Reflections on group work and sources

1.a Who are the members of your group that participated in the assignment.

Udover mig selv, har jeg udarbejdet opgaven i gruppe med:
- Daniel Friis-Hasché - rcb933
- Aksel Mannstaedt Rasmussen - qfl561

1.b Describe how you organize group work for the assignment and reflect on the quality of the group interactions.

Ifølge min oplevelse, har gruppearbejdet fungeret godt. Vi har snakket meget om opgaverne, og vi har hjulpet hinanden, med at finde svar på spørgsmål. Vi har ikke haft meget direkte sammerbejde i opgaverne, men har i stedet delt vores opfattelse af opgaverne.

Den største grund til at vi ikke har arbejdet sammen mere, er fordi vi startede på afleveringen på forskellige tidspunkter. Det har derfor været svært, at tale om noget, de andre i gruppen ikke har beskæftiget sig med endnu. Jeg har derfor selv valgt, at starte på afleveringen tidligere end de andre, da jeg har haft mere tid til gode

1.c Give a list of the external sources you used during the assignment and how you used them.

Jeg har benyttet Claude 3.5 Sonnet til at genere koden til opgave 3.a:

```
let rec headInsert (h: int) (t: int list): int list =
    match t with
    | [] -> [h]
    | head :: tail when (h <= head) -> h :: t
    | head :: tail -> head :: (headInsert h tail)

let rec isortRec (l: int list): int list =
    match List.length l with
    | 0 | 1 -> l
    | _ -> headInsert l.Head (isortRec l.Tail)
```

https://claude.ai — Dato: 30/10/2024

Jeg har også benyttet Claude 3.5 Sonnet til at genere noget af koden i opgave 3.c:

```
let rec isortIter' (xs: int list) (ys: int list): int list =
    match ys with
    | [] -> xs
    | h :: t -> isortIter' (headInsert h xs) t
```

https://claude.ai — Dato: 01/11/2024

Alt den generede kode er derefter blevet redigeret og verificeret af mig, og resten af afleveringen er skrevet af mig selv i samarbejde med min gruppe.

2 2D vectors as a module

A vector space is a set V together with operations $+$ (addition), $*$ (scalar multiplication), $-$ (unary negation) and 0 (zero element) that satisfy a number of universally quantified properties; see "Vector space" on Wikipedia. (Note that scalar multiplication is often just written by juxtaposition in mathematics, without an explicit $*$; e.g. kx instead of $k * x$.)

2.a Define the abstract type of 2D vectors over floats (64-bit floating-point numbers) as an F# signature file Vec2D.fsi that contains:

```
module Vec2D

V = float * float

add : V -> V -> V //+
neg: V -> V //unary -
scale: float -> V -> V //*, scalar multiplication
zero: V //0
```

Add the properties of a vector space as a comment (* ... *) at the bottom of the signature file.

Jeg har defineret en abstract data type, præcist som den vist i opgave beskrivelsen. Denne type er en tuple, bestående af 2 floats, som skal repræsentere vektorkoordinater. Value'en for: add, neg, scale og zero er også defineret, som vist i opgaven.
Jeg har derudover lavet en kommentar i bunden af min Vec2D.fsi fil, der fortæller hvordan disse values skal fortolkes i forbindelse med vektorberegning:

```
module Vec2D

type V = float * float
```

```

val add: V -> V -> V           // +
val neg: V -> V                // unary -
val scale: float -> V -> V     // *, scalar multiplication
val zero: V                    // 0
(*
add = (x1 + x2, y1 + y2)
neg = (-x, -y)
scale = (k * x, k * y)
zero = (0, 0)
*)
```

2.b Define an implementation of module Vec2D and put it in file Vec2D.fs.

Jeg har tilføjet følgende til module'et Vec2D i implementation file'en:

```
module Vec2D

type V = float * float

let add ((x1, y1): V) ((x2, y2): V): V =
    ((x1 + x2), y1 + y2)

let neg ((x, y): V): V =
    (-x, -y)

let scale (k: float) ((x, y): V): V =
    (k * x, k * y)

let zero: V =
    (0.0, 0.0)
```

Her har jeg benyttet let-bindings til at definerer funktionerne for:
add, neg, scale og zero

Hver funktion benytter vores abstract data type 'V' (float ∗ float), og følger den tidligere givet valuation.

2.c Consider distributivity, one of the properties a vector space must have:

`for all k in float, x in V, y in V, k * (x + y)= (k * x)+ (k * y)`

Define a function:

`isDistributive: (k: float)-> (x: V)-> (y: V)-> bool`

such that isDistributive c v w returns true if and only if distributivity holds for the three input values c v w passed to isDistributive. Put it in the F# script file Vec2DTest.fsx.

Gennem let binding har jeg defineret en ny funktion i Vec2DTest.fsx, der ser således ud:

```
let isDistributive (c: float) (v: V) (w: V): bool =
    // k * (x + y) = (k * x) + (k * y)
    if (scale c (add v w) = add (scale c v) (scale c w)) then
        true
    else
        false
```

Her er vores inputs:
`(c : float) (v : V) (w : V)`
eller med andre ord, en konstant, som er af type float, og 2 vektoerer, der hver består af vores ADT 'V'. Ide'en bag funktion er, at replikere den givne property af distributivity i form af en ligning. Hvis siderne er lig hinanden, retunerer funktionen true, ellers retunerer den false.

2.d Design a test suite for distributivity, employing specification-driven testing using input partitioning. Note that the set of possible inputs to consider consists of all triples (c, v, w) where c is a float and each of v, w is a pair of float. Add the code and test data making up the test suite to file Vec2DTest.fsx. Document in your report how you came up with the test suite systematically.
Argue why you believe your test suite is at least as likely to result in a specification violation compared to other suites of the same size. Discuss how your design of the test suite relates to specification-driven testing using input partition as presented on the PoP lecture slides on higher-order functions and testing; that is, how it applies or differs from the description of input partitioning and its illustration there.

Ved brug af vores isDistributive funktion, kan vi teste, om forskellige inputs opholder distributivity for vektorer.
Vi vil nu lave en funktion, der implementerer isDistributive funktionen på valgte inputs, og som fortæller os, om disse inputs er gyldige for isDistributive, eller om de giver en error:

```
let test (c: float) (v: V) (w: V) =
    match isDistributive c v w with
    | true -> Ok (c, v, w)
    | _ -> Error (c, v, w)
```

Som vi kan se, indtager funktionen argumenterne c, v og w, hvor c er af type float, og v og w er af type V. Funktionen retunerer enten Ok- eller Error af (c, v, w).
Ide'en for funktionens struktur kom fra en lignende funktion, der blev gennemgået i forelæsningen i uge 7.
Jeg kan ikke advokere for at min funktion er mindst så god som andre suites, da jeg ikke har arbejdet eller set andre suites.

2.e Run the test suite to test whether the implementation of Vec2D does not satisfy distributivity. Does the test suite yield a specification violation or not?
Discuss:
- If it does result in a specification violation, is module Vec2D a vector space? If necessary, make changes to the comments in the Vec2D.fsi to reflect that your specification is consistent with your implementation. (Recall that whatever property is promised to hold in an .fsi file must be correctly implemented in the .fs file.)
- If it does not result in a specification violation, can you conclude that, assuredly, your implementation satisfies the distributivity property for all possible values for k, x and y? Justify your answer.

Vi kører vores test funktion på nogle normale og ekstreme eksempler:

```
printfn "%A" (test 2.0 (1.0, 1.0) (2.0, 2.0))
printfn "%A" (test 0.0 (-1.0, 1.0) (0.0, 2.0))
printfn "%A" (test System.Double.NaN (1.0, 1.0) (2.0, 2.0))
printfn "%A" (test System.Double.PositiveInfinity (1.0, 0.0) (0.0,
    1.0))
(* prints:
Ok (2.0, (1.0, 1.0), (2.0, 2.0))
Ok (0.0, (-1.0, 1.0), (0.0, 2.0))
Error (nan, (1.0, 1.0), (2.0, 2.0))
Error (infinity, (1.0, 0.0), (0.0, 1.0))
```

Vi ser, at vi får en error, når vi tester inputs'ne: NaN og infinity, som vores k-parameter.

Jeg kan ikke se, hvorfor infinity, ville give en specification violation, da jeg mindes, at vi til forelæsningen i uge 5, testede netop dette, uden det resulterede i en error.
Jeg tænker derfor, at NaN og infinity måske ikke opholder reglen for distributivity i vores isDistributive funktion, og dette resulterer i en error.

2.f Construct a project file Vec2D.fsproj that contains the final versions of your signature, implementation and script files above such that dotnet run executes and prints the results of the tests in Vec2DTest.fsx.

Jeg har skrevet følgende i min .fsproj file i Sort-mappen:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Vec2D.fsi" />
    <Compile Include="Vec2D.fs" />
    <Compile Include="Vec2DTest.fsx" />
  </ItemGroup>

</Project>
```

Derefter kan jeg køre den tidligere viste kode med dotnet build, og derefter dotnet run.

3 Sorting

A practically useful purely functional sorting algorithm is recursive insertion sort. It works like this:
Lists of length 0 or 1 are already sorted. Given a list of $n >= 2$ values $[x1; x2; \ldots; xn]$, first sort its tail $[x2; \ldots; xn]$ and then insert x1 into the sorted tail at the correct position. For example, if the input list is $[4; 2; 9; 3; 0]$, it is first pattern matched into the head of the list, 4, and its tail, $[2; 9; 3; 0]$. Then the tail is sorted, which yields $[0; 2; 3; 9]$. Finally, the head is inserted at the right position, resulting in $[0; 2; 3; 4; 9]$. (Insertion sort has a bad worst-case complexity of $O(n^2)$, but is practically useful since it is very efficient on 'almost-sorted' inputs.)

3.a Create file Sort.fs and add the definition of function:

`isortRec: int list -> int list`

which implements recursive insertion sort. (The definition must not use mutable variables, arrays, or other operations that have imperative side-effects.)

Employ the 8-step method to identify any auxiliary functions that may be required.

Kens method:

1. **Write a brief description of what the function should do**
2. **Find a name for the function**
3. **Write down test examples**
4. **Find out the type of inputs and outputs**
5. **Generate code for the function (and possibly helper functions)**
6. **Write test cases**

7. Write short documentation for the fucntion

Vores isortRec funktion bør indtage en liste, bestående af integers (type int), og outputte en (muligvis) ny int liste, der har de samme elementer, men sorteret. Dette skal gøres iterativt med recursion.
Eksempel vis, bør funktion kunne indtage følgende lister:

```
[9]
[]
[-1; -3]
[7; 1; 13; 0; -1]
```

Og outputte følgende sorterede lister:

```
[9]
[]
[-3; -1]
[-1; 0; 1; 7; 13]
```

Nedenunder har jeg defineret en auxiliary function, samt isortRec ved brug af let-bindings:

```
let rec headInsert (h: int) (t: int list): int list =
    match t with
    | [] -> [h]
    | head :: tail when (h <= head) -> h :: t
    | head :: tail -> head :: (headInsert h tail)

let rec isortRec (l: int list): int list =
    match List.length l with
    | 0 | 1 -> l
    | _ -> headInsert l.Head (isortRec l.Tail)
```

3.b Add to Sort.fs a function:

`isortRec' int list -> int list`

that is equivalent to isortRec. It should use List.foldBack instead of being recursively defined. A closely related sorting algorithm is iterative insertion sort. It works like this: Given an already sorted sublist $[x1; \ldots; xn]$ and a remaining unsorted sublist $[y1; y2; \ldots.; ym]$, $m >= 1$, insert $y1$ at the correct position in $[x1; \ldots; xn]$, which extends the already sorted sublist with one element, and then continue sorting $[y2; \ldots; ym]$. For example, on input $[4; 2; 9; 3; 0]$, after a couple of steps the already sorted sublist consists of $[2; 4; 9]$ and the list of remaining (unsorted) elements of $[3; 0]$. In the next step its head, 3, is inserted into the already-sorted sublist, which yields $[2; 3; 4; 9]$, and the sublist of remaining elements is $[0]$. After another step the already-sorted sublist contains $[0; 2; 3; 4; 9]$. Since the remaining list is empty, this is the final result.

I følgende har jeg lavet en identisk version af isortRec, men som benytter List.foldBack i stedet for recursion:

```
let isortRec' (l: int list) =
    match List.length l with
    | 0 | 1 -> l
    | _ -> List.foldBack headInsert l []
```

3.c Define function isortIter':

`(xs: int list)-> (ys: int list)-> int list`

that implements iterative insertion sort, where xs is the already sorted sublist, ys is the sublist of remaining elements, and the result is the final sorted list.

Define function isortIter:

`int list -> int list`

by

`let isortIter ys = isortIter' [] ys`

Add both isortIter' and isortIter to Sort.fs.

Her er koden til både isortIter og dens auxiliary function isortIter':

```
let rec isortIter' (xs: int list) (ys: int list): int list =
    match ys with
    | [] -> xs
    | h :: t -> isortIter' (headInsert h xs) t

let isortIter (ys: int list): int list =
    isortIter' [] ys
```

3.d Try out isortIter and isortRec on a number of examples and add the code to a file named SortExamples.fsx. (You are welcome to specify properties a sorting function has to satisfy and design a test suite, if you have time on hand. Designing a test suite systematically is not required here, however, only trying out the functions on some illustrative inputs.)

Her er nogle test examples, som jeg også nævnte tidligere, at funktionerne bør kunne håndtere:

```
printfn "%A sorted = %A" [9] (isortRec [9])
printfn "%A sorted = %A" [] (isortRec [])
printfn "%A sorted = %A" [-1; -3] (isortRec [-1; -3])
printfn "%A sorted = %A" [7; 1; 13; 0; -1] (isortRec [7; 1; 13; 0;
    -1])
(* prints:
[9] sorted = [9]
[] sorted = []
[-1; -3] sorted = [-3; -1]
[7; 1; 13; 0; -1] sorted = [-1; 0; 1; 7; 13]
*)

printfn "%A sorted = %A" [9] (isortIter [9])
printfn "%A sorted = %A" [] (isortIter [])
printfn "%A sorted = %A" [-1; -3] (isortIter [-1; -3])
printfn "%A sorted = %A" [7; 1; 13; 0; -1] (isortIter [7; 1; 13; 0;
    -1])
(* prints:
[9] sorted = [9]
[] sorted = []
[-1; -3] sorted = [-3; -1]
[7; 1; 13; 0; -1] sorted = [-1; 0; 1; 7; 13]
*)
```

3.e Define an imperative implementation of iterative insertion sort, using arrays. The same array is used to store both the already sorted and the yet-to-be-sorted elements. Specifically, define a non-recursive function `isortImp: int array -> unit`
that inputs an array of integers and updates it such that, upon returning, the array contains the same elements, but in ascending order. The implementation of isortImp must not use lists and use only one array, the input array, which is updated imperatively; it must not use recursion, but use control flow constructs such as for- or while-loops instead. No new arrays must be created.. Add isortImp to Sort.fs and add illustrative examples of its use to SortExamples.fsx.

Nedenunder har jeg defineret isortImp:

```
let isortImp (A: int array): int array =
    let n = Array.length A
    let mutable e = 0
    for i = 0 to n - 1 do
        for j = n - 1 downto i + 1 do
            if A[j] < A[j - 1] then
                e <- A[j]
                A[j] <- A[j - 1]
                A[j - 1] <- e
    A
```

Her har jeg benyttet et bubblesort algorithm, som imperativt benytter arrays og variablen e til at opbevare elementer. Disse elementer bliver gennem iteration i for-loops sorteret, hvor e benyttes til at reassigne værdier i den rigtige rækkefølge.

3.f Construct a project file Sort.fsproj that contains the implementation and script files above such that dotnet run executes and prints the results of all the examples in SortExamples.fsx.

Min .fsproj fil ser således ud:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Sort.fs" />
    <Compile Include="SortExamples.fsx" />
  </ItemGroup>

</Project>
```