Ugeaflevering 6g

Niels Abildskov (vzn250), Niklas Marcussen (gwv160)

4. november 2023

1 Introduktion

Denne rapport omhandler implementering af *specification-driven testing* hvilket benyttes til at opstille en *text suite* for kø-strukturer. Suiten testes imod en implementering som givet i opgavebeskrivelsen. Envidere bliver der i rapporten implementeret forskellige funktioner til manipulation af binære søgetræer, og nogle af disse vil ligeledes afprøves ved *specification-driven testing*. Afsluttende diskuteres livsagtige cellulære automater og der fremlægges kode til at implementere sådanne, samt til at vise dem grafisk vha. DIKU Canvas[1]

2 FastQueue

I denne sektion fokuserer vi på problemanalysen og designet af en immutable queue der er specificeret som en abstrakt datatype (ADT). Denne ADT er repræsenteret som følgende:

```
type Queue<'a>
emptyQueue : Queue<'a>
enqueue: Queue<'a> * 'a -> Queue<'a>
dequeue: Queue<'a> -> ('a * Queue<'a>) option
fromList: 'a list -> Queue<'a>
toList: Queue<'a> -> 'a list
```

Egenskaber

1. toList (fromList xs) = xs for en arbitrær liste, xs
2. toList emptyQueue = []
3. toList (enqueue (fromList xs, xs)) = xs @ [x]
4. dequeue emptyQueue = None
5. dequeue (fromList (x :: xs)) = Some (x, q) og toList q = xs

Ud fra egenskaberne beskrevet af *FastQueue* er vi kommet frem til følgene ækvivalensklasseopdelinger:

1. enqueue
   (a) emptyQueue
   (b) Non emptyQueue
2. dequeue
   (a) emptyQueue
   (b) Non emptyQueue
3. fromList
   (a) Empty input-list

[1] `https://github.com/diku-dk/diku-canvas`

(b) Non empty input-list

4. toList

   (a) Empty queue

   (b) Non empty queue

2.1 Programbeskrivelse

For at udføre de test-cases vi har opstillet i vores problemanalyse har vi defineret en række funktioner som vi har kaldt test-properties, hver af disse test-properties funktioner vil udføre en af vores test-cases. I de fleste tilfælde har vi blot lavet en if-else statement der ser om vores test-properties får det rigtige output af den pågældende funktion i *FastQueue*.

I test-property5 hvor vi tester om *dequeue* af en ikke *emptyQueue* returnere det korrekte element og den korrekte liste. Dette gør vi således:

```
let test_property5 =
    let x = 1
    let xs = [2; 3; 4]
    let q = fromList (x :: xs)
    match dequeue q with
    | Some (resultX, resultQ) when resultX = x && toList resultQ = xs ->
        printfn "Property 5: Passed"
    | _ ->
        printfn "Property 5: Failed"
```

Denne test-property funktion konverterer en liste *xs* til en queue igennem *fromList* og derefter kalder den *dequeue* med den nylavede queue *q*. Herefter matcher den de returnerede elementer fra *dequeue* med det forventede output. Hvis *dequeue* fungere korrekt vil funktionen printe "*Property* 5 : *Passed*.

2.2 Afprøvning og Konklusion

Vi har i nedenstående defineret et set test-cases ud fra vores ækvivalensklasseopdelinger, som vil sikre, at en eventual implmentation er korrekt ift. de angivne egenskaber

- At fromList og toList kun ændrer på typen, og ikke dataen, listen indeholder (jf. egenskab 1)
- At toList af en emptyQueue vil resultere i en emptyList (jf. egenskab 2)
- At enqueue tilføjer et element bagerst i en queue (jf. egenskab 3)
- At dequeue returnere None hvis kaldt på en emptyQueue (jf. egenskab 4)
- At dequeue popper det forreste element i en queue, men beholder resten (jf. egenskab 5)

Igennem disse tests kan vi teste alle de påkrævede egenskaber og derved kan vi finde ud af om det er en korrekt implimentering af *FastQueue*. *FastQueue* implimentationen har bestået 5/6 af vores test-cases. fromList, toList og enqueue fungerere alle som forventet og uden problemer. Dog i vores test-property4 hvor vi tester *dequeue* på en *emptyQueue*, returnerer koden ikke *None* som var forventet. Dette er fordi i implimentationen af *FastQueue* er dequeue defineret som følgende:

```
let rec dequeue ((frontq, endq) : Queue<'a>) : ('a * Queue<'a>) option =
    match frontq with
    | [] -> dequeue (List.rev endq, [])
    | x :: xs -> Some (x, (xs, endq))
```

problemet med denne funktion er at hvis *dequeue* bliver kaldt med en *emptyQueue* vil den skabe et uendeligt rekursivt loop som stammer fra det første match-pattern. Den vil kalde sig selv med en reversed Queue, og dette vil den gøre indtil vi får en SOE[2].
Dermed kan vi konkludere at denne implimentation af *FastQueue* ikke opfylder alle de givne egenskaber ud fra vores test-cases der tjekker alle vores ækvivalensklasser.

[2]Stack-Overflow Exception

3 Binary Tree

I den næste del af vores rapport fokuserer vi på binære træer og deres tilsvarende ADT. En binær trætype (`BTree<'a>`) er specificeret med to typer: `Leaf` (tomme blade) og `Branch`, der har en værdi 'a samt venstre og højre undertræer.
Vi skal lave en funktion, *size*, der finder antallet af knuder i vores træ. For at gøre dette skal vi tage summen af knuderne i alle subtrees på både venstre og højre side og ligge 1 til for vores initalknude. Derudover skal vi lave en funktion, *fold*, der udfører en specifik funktion på hver knude i træet. Dette kan vi gøre ved rekursivt udføre en funktion for hver knude i alle subtrees. Vi er kommet frem til følgende ækvivalensklasseopdelinger til $BTree$

1. size
   (a) emptyTree som input
   (b) Single node tree som input
   (c) Multiple node tree som input
2. folder
   (a) emptyTree som input
   (b) Single node tree som input
   (c) Multiple node tree som input

3.1 Programbeskrivelse

3.1.1 Funktion size

Vores funktion, *size*, kører rekursivt igennem alle subtrees på venstre og højre side og finder summen af knuderne + 1 for vores intialknude. Dette gør den således:

```
let rec size tree =
match tree with
| Leaf -> 0
| Branch (left, _, right) -> 1 + size left + size right
```

3.1.2 Funktion fold

Vores funktion, *fold*, kører ligesom *size* rekursivt igennem alle subtrees på venstre og højre side, men i modsætning til *size* kører den en funktion, *folder*, som tager værdien fra venstre og højre undertræ, samt den nuværende knudes værdi. fold implementeres således

```
let rec fold (folder: 'b  *'a * 'b -> 'b)
             (acc: 'b) (tree: BTree<'a>) : 'b =
    match tree with
    | Leaf -> acc
    | Branch (l, v, r) ->
        let leftResult = fold folder acc l
        let rightResult = fold folder acc r
        folder (leftResult, v, rightResult)
```

Afsluttende definerer vi følgende funktion i 6gTests.fsx,

```
let folder acc tree =
    fold (fun (sizeLeft, x, sizeRight) -> sizeLeft + 1 + sizeRight) acc tree
```

Der returnerer (size tree) + acc

3.2 Afprøvning og Konklusion

For at teste denne implimentation af vores *size* og *fold* funktioner har vi udformet 3 test-cases, navnlig emptyTree, t1, og t5. De defineres således:

```
let emptyTree = Leaf
let t1 = Branch (Leaf, 1, Leaf)
let t2 = Branch (t1, 2, t1)
let t3 = Branch (Leaf, 3, Leaf)
let t4 = Branch (t3, 4, t3)
let t5 = Branch (t2, 5, t4)
```

Vi kalder vores *size* funktion på både et *emptyTree*, $t1$ (et træ med en knude) og på $t5$ (et træ med flere knuder), begge resultater bliver godkendt. Vores *folder* funktion bliver testet på et *emptyTree*, på $t1$ og $t5$, alle resultater bliver godkendt. Hermed har vi testet alle vores ækvivalensklasser.
Dermed kan vi konkludere at vores implimentation af *size* og *fold* fungerer korrekt.

4 Difference List

I den sektion af rapporten, fokuserer vi på en type difference list, som er repræsenteret i DiffList-modulet. Dette modul arbejder med lister og har forskellige operationer, såsom `fromList`, `toList`, `nil`, `cons`, og `append`. Vi ønsker at lave en funktion inorderD : BTree<'a> -> 'a dlist som returnerer en *inorder traversal* af et binært træ som en difference list. Dette gøres ved, at man ved en arbitrær knude, kalder funktionen rekursivt på venstre undertræ, tilføjer den aktuelle knudes værdi, og derefter prepend'er resultatet fra et rekursivt kald til højre undertræ. Da input-typen er det samme som for tidligere tests (size/fold), er ækvivalensklasseopdelingerne de samme, navnlig emptyTree, et træ med en knude, og et træ med flere knuder.

Funktioner og Typer

Vi er blevet givet følgende interface

```
module DiffList
type 'a dlist = 'a list -> 'a list
let fromList : 'a list -> 'a dlist = (@)
let toList (dl : 'a dlist) : 'a list = dl []
let nil : 'a dlist = fun ys -> ys    // = fromList []
let cons (x : 'a, dl : 'a dlist) : 'a dlist =
    fun ys -> x :: dl ys
let append : 'a dlist -> 'a dlist -> 'a dlist = (<<)
```

4.1 Programbeskrivelse

Vi har lavet en rekursiv funktion *InorderD* der kører igennem alle knuder i alle subtrees på venstre og højre side, og gemmer deres værdier i en liste:

```
let rec inorderD (tree: BTree<'a>) : 'a dlist =
    match tree with
    | Leaf -> nil
    | Branch (left, value, right) ->
        let leftD = inorderD left
        let rightD = inorderD right
        leftD << (fun ys -> value :: rightD ys)
```

Vi ønsker nu, at definere en funktion inorder : BTree<'a> -> 'a list, og vi bemærker at dette kan gøres ved at kombinere toList og inorderD vha komposition på følgende måde:

```
let inorder (tree: BTree<'a>) : 'a list = (toList << inorderD) tree
```

4.2 Afprøvning og Konklusion

Vi tester $DiffList$ med en $emptyQueue$, en $Singlenodequeue$ og en $Multiplenodequeue$, disse test-cases er vi kommet frem til ud fra vores ækvivalensklasseopdeling.
Vores module $DiffList$ består alle vores tests og dermed er det en velfungerende implimentation af $DiffList$.

5 Cellulære automater

5.1 Problemstilling

Vi ønsker en implementering af cellulære automater. Til at opnå dette formål, kan 8-trins vejledningen[3] til at designe funktioner benyttes. Navn og interface/type er givet i opgavebeskrivelsen, så disse udelades fra nedenstående overvejelser

- **Ekstern test:** Da det ikke er del af opgaven at definere tests til de cellulære automater, vil der ikke blive skrevet kode til det formål, men ved at håndkøre eksempler ville man kunne teste Update<'a> funktionen mod godkendte resultater. Dette ville være en form for specification-driven testing, dog med kendskab til funktionen, der står for at bestemme en celles nye værdi.
- **Implementering:** Dette er beskrevet mere detaljeret i sektion 5.2, men update-funktionen opdeles i to dele. En til at finde naboer til en given celle, og en til at tage disse, samt det nuværende celles værdi, og pipe det ind i reglen, brugeren har angivet.
- **Dokumentation:** Vi benytter her XML-dokumentation på funktionerne.

5.2 Programbeskrivelse

Opdateringsfunktionen for en cellulær automat, givet en størrelse og en opdateringsregel, vil - for hver celle - skulle udføre følgende operationer:

1. Find dens gyldige naboer (Moore neighbourhood). Altså naboer, som er indenfor gitteret.
2. Ud fra disses værdier, samt cellens egen værdi, skal rule-funktionen kaldes for at få den nye generations værdi.

Ovenstående kan opnås vha. hjælpefunktionen neighbourValues : Size -> State<'a> -> Pos -> 'a list. Denne skal lave en liste af nabo-kandidater[4], og derefter filtrere den med prædikatet $(x \geq 0 \wedge x < S_x) \wedge (y \geq 0 \wedge y < S_y)$ hvor (x, y) er en arbitrær nabo-kandidat, og $S = (S_x, S_y)$ er størrelsen af gitteret, som starter fra 0. Vi bruger følgende kald til List.filter for at få de gyldige naboer

```
let predicate ((x,y) : Pos) : bool
    = (x >= 0 && x < cols) && (y >= 0 && y < rows)
([
    (x-1,y-1);
    (x,y-1);
    (x+1,y-1);
    (x-1,y);
    (x+1,y);
    (x-1,y+1);
    (x,y+1);
    (x+1,y+1)
] |> List.filter predicate )
```

Hvorefter der bruges forward-piping til List.map (fun p -> state |> Map.find p) for at få naboernes værdier. Vi kan nu definere funktionen update : Size * Rule<'a> -> State<'a> -> State<'a> inde i cellularAutomaton

```
let update ((size, rule): Size * Rule<'a>) (state: State<'a>) : State<'a> =
    state |> Map.map (fun (pos : Pos) (value: 'a) ->
        neighbourValues size state pos |> rule value)
```

[3] Sporring, Learning to Program with F# p. 84 (2022-11-29)
[4] De otte felter rundt om en celle.

Dette lader os returnere settings |> update, som har signaturen State<'a> -> State<'a> (altså Update<'a>), fra cellularAutomaton.

5.3 Afprøvning

For at visualisere funktionaliteten fra sektion 5.2 implementeres der et interaktivt Canvas-program. Da dette ikke er en central del af opgaven (at implementere cellulære automater) udelades meget af beskrivelsen her[5], men for en kort bemærkning udføres følgende operationer:

1. Over hele vinduet vises en hvid rektangel, og gitter-linjer beregnes på baggrund af vinduets størrelse og antal celler.
2. Der defineres en event-handler, som skifter en celles værdi, hvis der trykkes på den, eller kører update-funktionen hvis der trykkes på mellemrum.

I Figur 1 vises evolutionen efter fire generationer. Dette opnåes ved at starte med et tomt gitter, og klikke på felterne med musen så man får mønstret i Figur 1a. Derefter bruges mellemrums-tasten til at opdatere felterne.

(a) Oprindelig tilstand (b) Efter 1 opdatering (c) Efter 2 opdateringer (d) Efter 3 opdateringer

Figur 1: Evolution over 4 generationer af Conway's Game of Life

[5] Funktionerne til det er dog dokumenteret med XML i koden