

DANMARKS TEKNISKE UNIVERSITET



Bachelor Projekt

TITLE

Jonas Dahl Larsen (s205829)

6. marts 2024

Indhold

1	Introduktion	2
2	Fundamentale koncepter	2
2.1	Introduktion til Funksions Programmering	2
2.1.1	Typen	3
2.2	Signatur filer og implementerings filer	4
2.3	Overloading operatorer	4
2.4	Property Based Testing	4
3	Symbolske lignings udtryk	5
3.1	Tal mængder	5
3.1.1	Rationelle tal Mondul	6
3.1.2	Komplekse tal Mondul	7
3.1.3	Tal Mondulet	7
3.2	Ligningsudtryk Mondulet	8
3.2.1	Ligninger som træer	8
3.2.2	Infix og Prefix notation	8
3.2.3	Generering af Ligningsudtryk	8
3.3	Simplifikation af Ligningsudtryk	8
3.3.1	PBT af simplifikationen	8
3.4	differentiering af Ligningsudtryk	8

1 Introduktion

I 2023 valgte Danmarks Tekniske Universitet at anvende Python som et hjælpeværktøj i deres grundlæggende matematikkursus "01001 Matematik 1a (Polyteknisk grundlag)". Python er et af de mest anvendte programmeringssprog¹, kun overgået af to sprog, der primært bruges sammen til at udvikle hjemmesider. Derfor har Python, med sit dynamiske skrevne sprog og en række matematiske programudvidelser som SymPy², været et oplagt valg som programmeringssprog til det grundlæggende matematikkursus tilbudt af DTU.

Projektet vil undersøge, hvordan et funktionsprogrammeringssprog, kan gavne de studerendes forståelse af de grundlæggende matematiske koncepter. Formålet er at guide læseren gennem opbygningen af en række funktionsprogrammer baseret på grundlæggende universitetsmatematik og dermed illustrere anvendelser. Projektet beskriver en generel struktur til opbygning og anvendelse af et funktions programmeringsprogram. Der tages udgangspunkt i F#³, men beskrivelserne af programmerne vil også kunne anvendes i lignende funktionsprogrammeringssprog.

Rapporten begynder med at forklare nogle Fundamentale koncepter inden for funktionsprogrammering samt metoder til validering af programmerne.

2 Fundamentale koncepter

2.1 Introduktion til Funktions Programmering

Det er forventet af læseren har kendskab til programmering, der gives derfor kun en kort beskrivelse af syntax og notationen, så læser ikke bekendt med F# kan forstå de eksempler der løbende vil forekomme i rapporten.

$$f(n) = \begin{cases} 1 & n = 0 \\ n \cdot f(n-1) & n > 0 \\ \text{undefined} & n < 0 \end{cases} \quad (1)$$

Vi begynder derfor med at betragte funktionen for fakultet 1, et eksempel på en implementering i F# er givet i 5 som kan sammelignes med Python kode 2, da Python og pseudokode er næsten det samme.

```
1 // Fakultet i F#
2 let rec factorial n =
3     match n with
4     | 0          -> 1
5     | x when x > 0 -> x * factorial (x - 1)
6     | _         -> failwith "Negative argument"
```

Listing 1: Eksempel på Fakultet i F#

¹<https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>

²<https://www.sympy.org/en/index.html>

³[https://en.wikipedia.org/wiki/F_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/F_Sharp_(programming_language))

```

1 // Fakultet i Python
2 def factorial(n):
3     if n == 0:
4         return 1
5     elif n > 0:
6         return n * factorial(n - 1)
7     else:
8         raise ValueError("Negative argument")

```

Listing 2: Eksempel på Fakultet i Python

I F# anvendes `let` til at definere en ny variabel eller, i dette tilfælde, en funktion kaldet `factorial`. Næste nøgleord er `rec`, hvilket indikerer, at funktionen er rekursiv. Funktionen tager et input-argument n , og i linje 3 starter et match-udtryk. Her er n vores udtryk, og efter `with` begynder en række mønstre, som udtrykket forsøger at matche på, separeret med `|`.

Et match på et mønster er ikke det samme som `'=='` kendt fra andre programmeringssprog. I linje 4 forsøger den at tildele værdien af n til 0, og dette lykkes kun, hvis n er 0. Hvis n ikke er 0 og dermed ikke matcher linje 4, vil den forsøge at matche det næste mønster. Her står der kun x , da der ikke yderligere specificeres om netop x , vil det altid lykkes, og x bliver tildelt værdien af n svarende til $[x \mapsto n]$. Derefter skal betingelserne efter `when` være opfyldt. Hvis de er det, kan mønsteret matche på den givne linje. Derefter eksekverer den og returnerer koden efter `'→'`, hvor den samtidig har adgang til den nyligt tildelte værdi af x . Det sidste mønster på linje 6 anvender `'_'` som mønster. Dette betyder, at det kan matche alle udtryk, men vi er ikke interesseret i at anvende værdien. I dette tilfælde kan det tredje mønster på linje 6 kun køres, når n er negativ.

Bemærk hvordan en funktion i F# altid returnerer sidste kørte linje af en funktion. Svarende til man ikke skrev `return` i linje 4 og 6 af Python koden², men værdierne stadig blev returneret.

2.1.1 Typer

Alle funktioner har en type i F#, typen for `factorial`⁵ er $int \rightarrow int$. Det er ikke muligt at kalde funktionen med et argument der ikke er af typen int . Typen for funktionen skrives som *Factorial* : $int \rightarrow int$. Vi kan dermed formulere følgende omkring typer⁴:

$$f : T_1 \rightarrow T_2$$

$$fe : T_2 \iff e : T_1$$

Er en funktion kaldt med et argument der ikke matcher funktionen type, gives en fejlmeddelelse. Derudover kan en type også være bestående af en tuple af typer.

$$f : T_1 * T_2 * .. * T_n \rightarrow T_{n+1}$$

$$f(e_1, e_2, .., e_n) : T_{n+1} \iff e_1 : T_1 \wedge e_2 : T_2 \wedge .. \wedge e_n : T_n$$

En tuple som kun består af to typer, kaldes et par.

I F# er det ikke nødvendigt at anvende parenteser som i andre programmeringssprog. De vil

⁴s14 FPU F#

derfor kun anvendes hvor det er nødvendige, gennem rapporten. Når man laver en tuple er det nødvendigt. Givet en funktion $g : T_1 \rightarrow T_2 \rightarrow T_3$ betyder den tager to udtryk af typen T_1 og T_2 hvor evalueringen af funktionen giver T_3

2.2 Signatur filer og implementerings filer

En standard F# fil er lavet med .fs extension, denne fil indeholder alt den kode som er nødt til at kunne køre programmet. En implementerings fil kan have en signatur fil t med .fsi extension, denne fil indeholder en beskrivelse af de typer og funktioner i implementerings filen som er tilgængelige for andre filer. En signatur fil kan derfor bruges som et blueprint for andre der ønsker at anvende eller replikere implementerings filen. I andre programmerings sprog vil man anse funktionerne i signatur filen som værende "public" og de funktioner der ikke er i signatur filen, men er i implementerings filen som værende "private". I F# er det ikke muligt at lave en funktion "private" i en implementerings

2.3 Overloading operatorer

I F# er det muligt at overloade standard operatore, så man kan anvende dem på egne typer. Det vil igennem rapporten blive anvendt til at definere matematiske operationer på de typer som vi kommer til at bygge.

2.4 Property Based Testing

Property Based Test (PBT) er en teknik til at teste korrekthed af egenskaber som man ved altid skal være opfyldt. En PBT test genere en række tilfældige input til en funktion og tester om en egenskab er opfyldt. Hvis en egenskab ikke er opfyldt, vil PBT give et eksempel på en fejl. De matematiske studerende på DTU, begynder med at lære om logik. I den forbindelse lærer man at en udsagnslogisk formel er gyldig (tautologi) hvis den altid er sand. Der eksistere mange teknikker til at vise at en udsagnslogisk formel er gyldig, i DTU's matematiske kursus lærer man at anvende sandhedstabellen. De viser hvordan 2 er gyldig.

$$P \wedge (Q \wedge R) \iff (P \wedge Q) \wedge R \quad (2)$$

Vi vil også kunne anvende PBT til at undersøge om 2 er gyldig³.

```
1 #r "nuget: FsCheck"
2 open FsCheck
3
4 // proposition formula: bool -> bool -> bool -> bool
5 let propositional_formula P Q R =
6     (P && ( Q && R )) = ((P && Q) && R)
7
8 let _ = Check.Quick propositional_formula
```

Listing 3: PBT af 2, for at undgå shortcircuiting har begge sider af lighedstegnet omgivet af parenteser.

```
Ok, passed 100 tests.
```

Listing 4: Output 3

Check.Quick er en del af "FsCheck" biblioteket, den tager en funktion som argument, og generere en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnere "true" for alle input, vil testen lykkedes. Hvis funktionen returnere "false" for et input, vil testen fejle og give et eksempel på et input der fejlede. I 3 er der anvendt "Check.Quick" til at teste om 2 er gyldig. Funktionen "Check.Quick" returnere "Ok, passed 100 tests." hvilket indikerer at 2 er gyldig. Det vigtigt her at forstå dette ikke er det samme som at bevise at den er gyldig, der er en mindre sandsynlighed for ikke alle muligheder er blevet testet. I dette tilfælde kan vi regne sandsynligheden for hvor vidt alle kombinationer er testet. $P, Q, R \in \{True, False\}$, derfor er der $2^3 = 8$ mulige kombinationer for input til funktionen. Hvis vi antager at alle kombinationer er lige sandsynlige, er sandsynligheden for at alle kombinationer er blevet testet $1 - (\frac{7}{8})^{100} = 0.999998$. Derfor er det meget sandsynligt at 2 er gyldig.

Det vil generelt ikke være muligt at regne denne sandsynlighed, da vi senere vil anvende PBT til at teste funktioner der tager argumenter som ikke har et endeligt antal kombinationer. Dog vil det stadig give en god indikation hvorvidt en egenskab er overholdt. I nogle tilfælde vil det være en fordel at opskrive en PBT før implementeringen af en funktion som man ved skal overholde en egenskab, på den måde anvende Test Driven Development (TDD) ⁵ til at teste om ens egenskab forbliver overholdt, under implementering.

3 Symbolske lignings udtryk

Det ønskes at kunne repræsentere simple ligninger som en type i F#. Vil derfor gennemgå en del teori og funktion som er nødvendige for at kunne dette. Det vil give os et grundlæggende fundament for at kunne udføre matematiske evalueringer som differentiering i F#. Som de fleste andre programmer har F# kun float og int som kan repræsentere tal. Derfor vil vi begynde med at definere et monad som indeholder en type for tal. Tanke gangen her at gennemgå en opbygning af en måde at kunne repræsentere ligninger samt simplificere dem. Vi begrænset os selv til at kun have matematiske operationer som addition, subtraktion, negation, multiplikation og division.

3.1 Tal mængder

Vi begynder med opbygningen af et monad som kan repræsentere tal mængder. Typen for tal, består af tre konstruktører, for henholdsvis heltal, rationale tal og komplekse tal. Dog er mondulet lavet med henblik på at kunne udvides med flere typer af tal. Måden resten af programmet er lavet på, gør de eneste krav til tal er at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Samt at tallet inden for addition og multiplikation er associative. Dette gælder blandt andet ikke for en vektor, derfor vil vi senere betragte at udvide programmet med en type for vektorer. En udvidelse kunne være for reelle tal, som kan håndtere "floating point errors"⁶, men for ikke at komplicere programmet vil vi i denne opgave ikke betragte floats.

⁵https://en.wikipedia.org/wiki/Test-driven_development

⁶https://en.wikipedia.org/wiki/Floating-point_error_mitigation

3.1.1 Rationelle tal Mondul

Repræsentationen af rationale tal kan laves ved hjælp af danne et par af integers, hvor den ene integer er tælleren og den anden er nævneren.

```
1 type rational = R of int * int
```

Listing 5: Typen for rationelle tal

Nedestående er der givet en signatur fil for rational mondulet 6. i Implementerings filen overloades de matematiske operatoren, ved hjælp af de klassiske regneregler for brøker⁷.

```
1 module rational
2
3 [<Sealed>]
4 type rational =
5     static member ( ~- ) : rational -> rational
6     static member ( + ) : rational * rational -> rational
7     static member ( + ) : int * rational -> rational
8     static member ( - ) : rational * rational -> rational
9     static member ( - ) : int * rational -> rational
10    static member ( * ) : int * rational -> rational
11    static member ( * ) : rational * int -> rational
12    static member ( * ) : rational * rational -> rational
13    static member ( / ) : rational * rational -> rational
14    static member ( / ) : int * rational -> rational
15    static member ( / ) : rational * int -> rational
16    static member ( / ) : int * int -> rational
17
18 val make          : int * int -> rational
19 val equal         : rational * rational -> bool
20 val posetive      : rational -> bool
21 val toString     : rational -> string
22 val isZero       : rational -> bool
23 val isOne        : rational -> bool
24 val isInt        : rational -> bool
25 val makeRatInt   : rational -> int
26 val greaterThan  : rational * rational -> bool
27 val isNegative   : rational -> bool
28 val absRational  : rational -> rational
```

Listing 6: Signatur filen for rational mondulet

For at kunne sammenligne, men også for nemmere at undgå for store brøker, vil alle rationelle tal blive reduceret til deres simpleste form. Dette kan gøres ved at finde den største fælles divisor (GCD)⁸. Der udover er det vigtigt at være opmærksom på man ikke foretager nul division. Derfor vil implementerings filen kaste en "System.DivideByZeroException" hvis nævneren er eller bliver nul. Signatur filen indeholder en række funktioner som bliver anvendt af andre filer.

⁷https://en.wikipedia.org/wiki/Rational_number

⁸https://en.wikipedia.org/wiki/Greatest_common_divisor

3.1.2 Komplekse tal Mondul

3.1.3 Tal Mondulet

Vi har nu beskrevet en måde at kunne repræsentere bruger definere tal på ved brug af typer i F#. Det vil derfor være oplagt at have en type som indeholder alle de typer tal vi ønsker at kunne anvende i de matematiske udtryk vi er ved at opbygge. Fordelen ved at samle dem til en type er at vi kan lave en række funktioner blandt andet matematiske operationer som kan anvendes på alle type tal. Vi begynder med at definere en type for tal 7, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

```
1 type Number = | Int of int | Rational of rational
```

Listing 7: Typen for Number

Betragtes signatur filen for Number mondulet 8, ses det at der igen er defineret overloading af de anvendte matematiske operationer. Derudover er der defineret en række funktioner som kan anvendes på Number typen.

```
1 module Number
2
3 [<Sealed>]
4 type Number =
5     static member ( + ) : Number * Number -> Number
6     static member ( - ) : Number * Number -> Number
7     static member ( * ) : Number * Number -> Number
8     static member ( / ) : Number * Number -> Number
9     static member ( ~- ) : Number -> Number
10
11 val zero : Number
12 val one : Number
13 val two : Number
14 val isZero : Number -> bool
15 val isOne : Number -> bool
16 val isNegative : Number -> bool
17 val abs : Number -> Number
18 val greaterThan : Number -> Number -> bool
19 val tryMakeInt : Number -> Number
20 val toString : Number -> string
```

Listing 8: Signatur filen for Number mondulet

Ved implementeringen af de matematiske operationer, hvis der eksistere en konstruktør i Number, der repræsenterer en tal mængde hvor alle andre konstruktører er delmængder af denne mængde. Er det muligt at definere en enkelt funktion som kan udføre alle binære operationer. Som et eksempel er funktionen 9 givet, som tager to tal og en funktion i form af den ønskede binære operation som parameter. Funktionen vil derefter matche på de to tal og anvende den operation på de to tal.


```
1 // makeRational: Number -> rational
2 let makeRational a =
3     match a with
4     | Int x      -> make(x, 1)
5     | Rational x -> x
6
7 // operation: Number -> Number -> (rational -> rational -> ...
  rational) -> Number
8 let operation a b f =
9     f (makeRational a) (makeRational b) |> Rational
```

Listing 9: Number.operation funktionen

Det vil her til være oplagt på alle de matematiske operationer at anvende en funktionen til at forsøge at konvertere tal typen til den simpleste talmængde, som i vores tilfælde er heltal. Dette er gjort ved at anvende funktionen `tryMakeInt` på alle de matematiske operations overloadnings 10.

```
1 // tryMakeInt: Number -> Number
2 let tryMakeInt r =
3     match r with
4     | Rational a when isInt a -> Int (makeRatInt a)
5     | _ -> r
6
7 type Number with
8     static member (+) (a, b) = operation a b (+) |> tryMakeInt
9     static member (-) (a, b) = operation a b (-) |> tryMakeInt
10    static member (*) (a, b) = operation a b (*) |> tryMakeInt
11    static member (/) (a, b) = operation a b (/) |> tryMakeInt
12    static member (~-) (a)    = neg a |> tryMakeInt
```

Listing 10: Overloadnings funktionerne for Number

Dermed har vi et mondulet som kan repræsentere tal, samt udføre matematiske operationer på dem. Vi vil nu begynde at betragte hvordan vi kan anvende den i et lignings udtryk.

3.2 Ligningsudtryk Mondulet

3.2.1 Ligninger som træer

3.2.2 Infix og Prefix notation

3.2.3 Generering af Ligningsudtryk

3.3 Simplifikation af Ligningsudtryk

3.3.1 PBT af simplifikationen

3.4 differentiering af Ligningsudtryk