

Grundläggande Datalogi

Övningar

Ana Bove*

23 februari 2004

1 Mängder och induktion

1. Gör övningarna 2.3 och 2.5 i KP.
2. Formulera den bevisprincip för binära träd (övning 2.5 i KP) som motsvarar matematisk induktion för naturliga tal.
3. Bevisa följande påståenden genom att använda matematisk induktion:
 - (a) $\forall n \geq 1 \in \mathcal{N}. 1 + 3 + 5 + \dots + (2n - 1) = n^2$;
 - (b) $\forall n \geq 1 \in \mathcal{N}. 2 + 4 + 6 + \dots + 2n = n^2 + n$;
 - (c) $\forall n \geq 1 \in \mathcal{N}. 1 + 5 + 9 + \dots + (4n - 3) = n(2n - 1)$.

2 Relationer

1. Rita tre cirklar som skär varandra så mycket som möjligt. Cirklarna innehåller de reflexiva, transitiva och symmetriska relationerna. Ge exempel på relationer som ligger i vart och ett av de 7 områdena! Ge också exempel på en relation som ligger utanför alla cirklar, dvs en relation som varken är reflexiv, transitiv eller symmetrisk.

3 Funktioner

1. Varför är en partiell funktion en slags relation? Ge exempel på en binär relation som inte är en partiell funktion.
2. Rita tre cirklar som skär varandra så mycket som möjligt. Cirklarna innehåller de partiella funktioner från \mathcal{N} till \mathcal{N} som är injektiva, surjektiva respektive totala. Ge exempel på funktioner som ligger i vart och ett av de 7 snitten! Ge också exempel på en funktion som ligger utanför alla cirklar, dvs en funktion som varken är injektiv, surjektiv eller total.
3. Gör övningarna 2.7–2.10 i KP.
4. Vad är $\text{Dom}_{(f \circ g)}$?

*Tack till Peter Dybjer och Bengt Nordström. Modifierad av Nils Anders Danielsson.

4 Uppräkneliga Mängder

(KP använder synonymen *uppräkningsbara*).

1. Börja med att göra övningarna 3.1–3.4 i KP.
2. Läs beviset för sats 3.4 i KP. Skriv sedan ett Haskellprogram

```
diagonalize :: (Nat -> Nat -> Nat) -> Nat -> Nat
```

som tar en uppräkning (dvs en total surjektiv funktion) f av totala funktioner på Nat (Haskells implementering av naturliga tal) och returnerar en total funktion som inte finns med i uppräknningen! Programmet kan ses som ett “bevis” för att mängden av totala funktioner på de naturliga talen inte är uppräknelig.

3. Här är ett “bevis” av att de naturliga talen inte är uppräknelige. Var är felet? Betrakta varje naturligt tal i dess binära representation, dvs som en lista av bitar. Låt $f_1, f_2, \dots, f_k, \dots$ vara en uppräkning av de naturliga talen. Skapa ett element d genom att låta dess i -te bit vara negationen till f_i -s i -te bit (om en sådan finns, annars låter vi den vara 1). Men d kan inte vara med i uppräknningen eftersom den skiljer sig från alla tal i uppräknningen i åtminstone en bit.
4. I ett diagonaliseringsargument konstruerar man en funktion som skiljer sig på en diagonal från andra funktioner. Varför tar man diagonalen? Varför inte gå rakt ner? Eller rakt åt höger? Eller två (i stället för ett) steg ner och ett till höger?
5. Kan man räkna upp alla totala funktioner från \mathcal{N} till $\{1\}$? Motivera!
6. Kan man räkna upp alla partiella funktioner från \mathcal{N} till $\{1\}$? Motivera!
7. Som alla vet, är ju $\text{List}(\mathcal{N})$ en uppräknelig mängd. Vad är felet i följande “bevis” av motsatsen?

Antag att $\text{List}(\mathcal{N})$ är uppräknelig. Då finns en uppräkning, en total surjektiv funktion $f \in \mathcal{N} \rightarrow \text{List}(\mathcal{N})$. Vi kan nu skapa en lista ℓ som inte finns med i uppräknningen på följande sätt: Vi ser till att ℓ skiljer sig från listan $f(0)$ genom att låta ℓ -s första element vara $f(0)$ -s första element plus ett. Om $f(0)$ inte har något första element låter vi ℓ -s första element vara 0. Vi fortsätter på samma sätt genom att låta ℓ -s element nummer i vara strikt större än $f(i)$ -s element nummer i om elementet existerar. Annars låter vi ℓ -s element nummer i vara 0.

Vi ser nu att listan ℓ inte kan finnas med i uppräknningen. Om den funnes skulle $\ell = f(j)$ för något j . Men ℓ är konstruerad så att ℓ -s j -te element skiljer sig från $f(j)$ -s j -te element. Vi har fått en motsägelse och alltså måste antagandet att $\text{List}(\mathcal{N})$ är uppräknelig vara felaktigt.

8. Är mängden av listor med element ur en uppräknelig mängd alltid uppräknelig? Motivera!
9. Visa att mängden av heltal är uppräknelig!

10. Visa att de (positiva) rationella talen är uppräknliga!
11. Visa att mängden av decimaltal (flyttal i decimal representation) är uppräknelig!

5 Ändliga Automater (ÄA)

1. Konstruera en ändlig automat som accepterar språket $\{0(10)^n | n \in \mathcal{N}\}$.
2. Konstruera ändliga automater för följande språk över alfabetet $\Sigma = \{a, b\}$:
 - (a) Mängden av strängar med högst två a .
 - (b) Mängden av strängar som innehåller precis två a .
 - (c) Mängden av strängar vars längd är högst 3.
 - (d) Mängden av strängar där två tecken som kommer efter varandra alltid är olika, dvs det får inte finnas några delsträngar på formen aa eller bb .
3. KP övning 6.5.
4. I allmänhet kan man implementera en given ÄA genom att definiera de ändliga typerna `Sigma` och `State`, starttillståndet

```
start :: State
```

övergångsfunktionen

```
delta :: (Sigma, State) -> Maybe State
```

samt den karakteristiska funktionen för mängden av sluttillstånd

```
final :: State -> Bool
```

Övergångsfunktionen är ju en partiell funktion, och man kan ofta implementera en partiell funktion `f` från `a` till `b` som en total funktion `f'` från `a` till `Maybe b`, genom att låta

```
f' a = Nothing
f' a = Just b
```

om `f a` är odefinierad respektive `b`. Det är därför som `delta` har typen ovan.

Implementera ändliga automater i Haskell. Interpretatorn (eller simulatorn) blir en algoritm för strängmatchning:

```
accept :: ((Sigma, State) -> Maybe State) ->
          (State -> Bool) -> State -> [Sigma] -> Bool
```

så att

```
accept delta final start string = True
```

om automaten med övergångsfunktion `delta`, sluttillståndstest `final` och starttillstånd `start` accepterar strängen `string`.

Representera några av de ÄA du använt ovan och testkör.

6 Ändliga Tillståndsmaskiner (ÄTM)

1. Implementera en ÄTM i Haskell. Betrakta t.ex. den som beskrivs med en graf överst på sid 193 i KP. Först behöver vi representera alfabetet Σ som en ändlig datatyp. Här är $\Sigma = \{0, 1\}$ så vi definierar

```
data Sigma = 0 | 1
```

(Skall vi köra det här i Haskell så får vi förstås byta namn på 0 och 1.) Sedan definierar vi tillståndsmängden **State** som en annan ändlig datatyp. Här är **State** = $\{S\}$ så vi definierar datatypen

```
data State = S
```

Implementera sedan övergångsfunktionen δ som en funktion

```
delta :: (Sigma, State) -> (Sigma, State)
```

2. Implementera på motsvarande sätt den ÄTM som adderar binära tal enligt grafen överst på sid 194! (Det finns ett tryckfel i bilden! Var? Se erratabladet för KP som kan fås från kurshemsidan.)
3. I allmänhet kan man alltså implementera en given ÄTM genom att definiera de ändliga typerna **Sigma** och **State**, starttillståndet

```
start :: State
```

samt övergångsfunktionen

```
delta :: (Sigma, State) -> Maybe (Sigma, State)
```

Ofta är det naturligt att skilja på indataalfabetet **SigmaIn** och utdataalfabetet **SigmaOut** (jfr diskussionen efter Definition 6.1 sid 192 i KP). I så fall får övergångsfunktionen i stället typen

```
delta :: (SigmaIn, State) -> Maybe (SigmaOut, State)
```

Programmera en interpretator (eller "simulator") för ÄTM

```
evalFsm :: ((SigmaIn, State) -> Maybe (SigmaOut, State)) ->  
State -> [SigmaIn] -> [SigmaOut]
```

så att

```
evalFsm delta start in = out
```

om **out** är den utdatasträng som en ÄTM med starttillstånd **start** och övergångsfunktion **delta** ger som resultat när den exekveras med indatasträngen **in**. Notera att **evalFsm** kan skrivas som en polymorf funktion:

```
evalFsm :: ((a, c) -> Maybe (b, c)) -> c -> [a] -> [b]
```

Implementera den gärna och kör några testexempel.

4. Gör KP övning 6.1 och 6.2. Implementera också dessa i Haskell och testa med hjälp av din simulator.

7 Primitiv/Strukturell Rekursion och (Primitiv) Rekursiva Funktioner

1. Gör övning 2.11–2.16 i KP som handlar om att programmera med primitiv rekursion.
2. Språket \mathcal{PRF} av *primitivt rekursiva funktioner* (avsnitt 2.9.4 i KP¹) kan betraktas som en delmängd av Haskell på följande sätt. Varje \mathcal{PRF} -uttryck är ett Haskellprogram som är en n -ställig funktion på de naturliga talen för $n \in \mathcal{N}$, dvs ett Haskellprogram av någon av typerna

```
Nat
Nat -> Nat
Nat -> Nat -> Nat
Nat -> Nat -> Nat -> Nat
...
```

Varje sätt att bilda \mathcal{PRF} -uttryck svarar mot en viss Haskellfunktion. Vi kallar denna Haskellfunktion en \mathcal{PRF} -kombinator. T.ex. har vi

```
zero :: Nat
succ :: Nat -> Nat
zero = Zero
succ n = Succ n
```

Problemet är att vi behöver oändligt många \mathcal{PRF} -kombinatorer för att kunna bygga upp alla \mathcal{PRF} -uttryck! T.ex. så behöver vi oändligt många projektionsfunktioner

```
proj_i^n :: Nat -> Nat -> ... -> Nat -> Nat

proj_i^n x1 x2 ... xi ... xn = xi
```

en för varje $n \geq 1$ och $1 \leq i \leq n!$ Förklara på motsvarande sätt vilka \mathcal{PRF} -kombinatorer man behöver för att koda komposition och primitiv rekursion! (Du kan ersätta `Nat` med `Int` när du programmerar i Haskell.)

3. Definiera additionsfunktionen i Haskell med hjälp av \mathcal{PRF} -kombinatorerna från övning 7.2 (se sid 23–24 i KP).
4. (svårt) Gör övning 2.18–2.19 i KP som handlar om programmering i \mathcal{PRF} .
5. Gör övning 2.24b och 2.26 i KP som handlar om att programmera med strukturell rekursion över listor.
6. Definera en funktion som konkatenerar (slår samman) två listor.
7. Gör övning 2.27 i KP.

¹Notera att `zero` är en konstant funktion av ett argument i KP, inte som här en funktion av noll argument.

8 λ -kalkyl

1. Gör KP 5.1–5.3.
2. Vilka är de fria och bundna variablerna i följande matematiska uttryck:

$$\sum_{i=0}^n (i + x)?$$

Formalisera detta uttryck med hjälp av λ -notation. Du får förutsätta att du har definierat en konstant `sum` (som implementerar \sum) och att du redan har definierat `add` som ovan!

3. Vilka är de fria och bundna variablerna i uttrycket $\{x \in y \mid x^2 < z\}$?
4. Ge en induktiv definition av abstrakta syntaxen för uttrycken i λ -kalkyl. Blir detta en uppräknelig mängd?
5. Man kan använda λ -notation för att formalisera predikatlogik genom att införa konstanter för de logiska operationerna `and`, `or`, `not`, etc. och även *kvantorerna*

```
all      :: (Nat -> Bool) -> Bool
exists  :: (Nat -> Bool) -> Bool
```

Vilka är de fria och bundna variablerna i uttrycket $\forall x. \exists y. x < y + z$? Formalisera detta uttryck med λ -notation och `all` och `exists`! Du behöver då också en konstant `lt` för att representera `<`.

6. Gör KP 5.5 och 5.8–5.9. Lösningsskriptet talar också om η -reduktion i övningar 5.8 och 5.9.
7. Är det sant att följande två uttryck är α -ekvivalenta för alla e ?

$$\lambda x. e \equiv \lambda y. e[x := y]$$

8. Låt $\omega = \lambda x. x x$ och $1 = \lambda y z. y z$. Hur kommer $\omega 1$ att reduceras?
9. Vad är normalformen för uttrycket $(\lambda x y. y x) y$?
10. Definiera en förenklad substitutionsoperation för λ -kalkyl. Värdet av funktionen `closedsubst(e, x, e')` skall vara resultatet av att substituera uttrycket e' för all fria förekomst av variabeln x i uttrycket e . Uttrycket e' skall vara slutet. Är uttrycket `closedsubst(e, x, e')` alltid öppet om e är öppet?
11. Använd kombinatorerna `true = $\lambda x y. x$` och `false = $\lambda x y. y$` för att representera sanningsvärden i ren λ -kalkyl. Då kan fallanalys implementeras med kombinatorn `if = $\lambda x y z. x y z$` . Skriv program för `and`, `or` och `not` i ren λ -kalkyl med hjälp av dessa.

12. Visa att λ -uttrycket

$$Z \stackrel{\text{def}}{=} \lambda f. ((\lambda x. f(\lambda z. x x z))(\lambda x. f(\lambda z. x x z)))$$

är sådant att $Z g = g(Z g)$ för alla g . (Man behöver använda \rightarrow_η för att bevisa detta.)

13. (svårt) Gör KP 5.17.

9 PCF

1. Skriv PCF-program för sanningsvärdesfunktionerna `and`, `or` och `not`!
2. Hur översätter man `let`-uttryck i Haskell till PCF (utan att utföra substitutionen)? T.ex. vad blir

```
let x = add 3 2 in mul 3 x
```

och vad blir

```
let x = add 3 1 ; y = minus 5 1 in mul y x?
```

3. Skriv ett PCF-program `compose` för funktionskomposition! Vilken typ skulle det ha i Haskell?
4. Skriv ett PCF-program

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

som tar en 2-ställig funktion och returnerar en 2-ställig funktion `f` som beter sig på samma sätt som `f` men med argumenten omkastade.

5. Definiera kombinatorerna

```
k :: a -> b -> a  
s :: (a -> b -> c) -> (a -> b) -> a -> c
```

i PCF. (Se KP 5.3.3 där de kallas **K** och **S**.)

6. Kan du skriva ett PCF-program för `sum` i övning 8.2?
7. Skriv PCF-program `add` för addition och `mul` för multiplikation av naturliga tal.
8. Skriv ett PCF-program `eq` för likhet mellan naturliga tal!
9. Skriv ett PCF-program `minus` för subtraktion där

```
minus m n
```

är odefinierat om $m < n$.

10. Skriv ett PCF-program för trunkerad subtraktion `monus`, dvs

```
monus m n = minus m n
```

om $m \geq n$ och 0 annars.

11. Skriv PCF-program `div` och `mod` för kvot och rest vid division av naturliga tal.
12. Skriv ett PCF-program `factorial` för fakultetsfunktionen.
13. Skriv ett PCF-program `fib` så att `fib n` är det n -te fibonaccitalet.

14. Skriv ett PCF-program `iter` så att

```
iter n f
```

returnerar `f` komponerat med sig själv `n` gånger, dvs

```
iter 0 f e = e
iter 1 f e = f e
iter 2 f e = f (f e), etc
```

I Haskell skulle `iter` ha typen

```
iter :: Nat -> (Nat -> Nat) -> Nat -> Nat
```

15. Skriv ett PCF-program `natrec` så att

```
natrec d e Zero      = d
natrec d e (Succ n) = e n (natrec d e n)
```

16. Definiera `iter` med hjälp av `natrec` men utan att använda `fix`!
17. Gör övningarna 2.11–2.16 i KP igen, men nu i PCF. Eftersom det handlar om primitiv rekursion så får ni inte använda `fix`, men däremot `natrec`. (2.15 är antagligen svårare än de övriga.)
18. Skriv ett PCF-program som implementerar μ -operatören (“minimalisering”) som definieras på sidan 210 avsnitt 6.3 i KP! Notera att vi återigen behöver oändligt många program, ett för varje ställighet n . Definiera den 1-ställiga och den 2-ställiga varianten:

```
mu_1 :: (Nat -> Nat) -> Nat
mu_2 :: (Nat -> Nat -> Nat) -> Nat -> Nat
```

Definiera funktionen

```
max :: Nat -> Nat -> Nat
```

med hjälp av minimalisering!

19. När vi beskriver PCF-semantiken, skriver vi $e \rightarrow e'$ om e kan reduceras till e' i godtyckligt antal steg. Vi säger också att \rightarrow är en binär relation. Mellan vilka mängder är den en relation? Är den reflexiv? symmetrisk? transitiv?
20. Skriv ut reduktionssekvenserna för PCF-programmen

```
not True
and True True
```

där `not` och `and` definierades i uppgift 9.1. Hur många steg behövs det för att reducera dem till kanonisk form?

21. Skriv ut reduktionssekvensen för PCF-programmet

```
add 1 1
```

där `add` definierades i uppgift 9.7. Hur många steg behövs det för att reducera det till kanonisk form?

22. Är `all` och `exists` från övning 8.5 “beräkningsbara” i den intuitiva meningen att de kan implementeras av PCF-program och ändå få avsett beteende?
23. (svårt) Visa att PCF kan översättas till ren λ -kalkyl! Vi behöver alltså översätta alla PCF-konstanter. I övningen 8.11 visades hur man gör med sanningsvärdena

```
True, False, if.
```

Läs sedan om Churchtal i föreläsninganteckningarna och övning 5.17 i KP. Där står det hur man definierar

```
Zero, Succ
```

Övning 5.21 i KP handlar om hur man definierar `pred` (svårt!). Avsnitt 5.3.5 i KP handlar sedan om hur man definierar `fix`. Återstår `isZero`. Hur definierar man den? Ledning: använd `iter` som definieras i övning 9.14.

10 Beräkningsbarhet

1. Skriv ner bevisen för sats 2–4 i PD 2.1.
2. Det finns ett utkast till beviset för sats 6 i PD 2.1. Skriv ner detta bevis i detalj. Ledning: visa att du kan definiera `isNat` om du har tillgång till `eqFun!`
3. (svårt) I PD 2.2 använder vi oss av följande “zick-zack-kodning” av naturliga tal (se också övning 3.2 i KP):

```
paircode :: Nat -> Nat -> Nat
```

```
paircode m n = (m + n + 1)(m + n)/2 + m
```

Skriv Haskellprogram

```
fstcode :: Nat -> Nat
```

```
sndcode :: Nat -> Nat
```

som givet en kod p för ett par av naturliga tal (m, n) returnerar m respektive n , dvs

```
fstcode (paircode m n) = m
```

```
sndcode (paircode m n) = n
```

Skriv sedan PCF-program för `fstcode` och `sndcode`.

4. Skriv Haskellprogrammen `funpart` och `argpart` som används för självinterpretatorn i 2.3. De tar en kod för en applikation (dvs ett tal $n \geq 10$) och returnerar funktionsdelen respektive argumentdelen. Dvs

```
n = 10 + (paircode (funpart n) (argpart n))
```

Ledning: använd `fstcode` och `sndcode` från föregående övning.

5. Skriv ut beviset för sats 8 i PD 2.4.
6. Visa att det finns oändligt många totala funktioner i $\mathcal{N} \rightarrow \mathcal{N}$ som är PCF-beräkningsbara.
7. Hur vet man att mängden av PCF-beräkningsbara funktioner i $\mathcal{N} \rightarrow \mathcal{N}$ är uppräknelig? Motivera!
8. Ett PCF-program f beräknar en total funktion på naturliga tal omm det för alla $m \in \mathcal{N}$ finns ett $n \in \mathcal{N}$ så att $f m \Rightarrow n$. Dvs f terminerar alltid med ett naturligt tal som utdata om det får ett naturligt tal som indata. Visa att det inte finns något PCF-program `isTotal` så att `isTotal f` \Rightarrow `True` omm f är en total funktion på naturliga tal och `isTotal f` \Rightarrow `False` annars.
9. Ett PCF-program f beräknar den helt odefinierade funktionen på naturliga tal omm det för alla $m \in \mathcal{N}$ inte finns något $n \in \mathcal{N}$ så att $f m \Rightarrow n$. Dvs f terminerar aldrig med ett naturligt tal som utdata om det får ett naturligt tal som indata. Visa att det inte finns något PCF-program `isUndef` så att `isUndef f` \Rightarrow `True` omm f beräknar den helt odefinierade funktionen på naturliga tal och `isUndef f` \Rightarrow `False` annars.

11 Turingmaskiner

1. Implementera en turingmaskininterpretator (simulator) i Haskell. Först definierar vi datatypen

```
data TmInstr a = Write a | L | R
```

för turingmaskininstruktioner. Sedan definierar vi turingmaskinens minne, en s.k. positionerad remsa, som en trippel

```
type Tape a = ([a], a, [a])
```

så att i

```
(as, a, bs)
```

a betecknar den symbol där turingmaskinens läshuvud befinner sig, bs betecknar listan av rutor till höger om läshuvudet och as betecknar listan av rutor till vänster om läshuvudet. Listan as innehåller remsans rutor i omvänd ordning, från höger till vänster, så att första elementet betecknar rutan just till vänster om läshuvudet.

Simulatorn blir då en funktion

```
evalTm :: ((b,a) -> Maybe (TmInstr a,b))
        -> b -> (Tape a) -> (Tape a)
```

så att

```
evalTm delta state intape = outtape
```

om turingmaskinen med övergångsfunktion `delta` och starttillstånd `state` och den positionerade remsan `intape` som indata ger den positionerade remsan `outtape` som utdata. Implementera `evalTm`!

Vidare behöver vi två funktioner

```
nat2tape :: Nat -> Tape Sigma
tape2nat :: Tape Sigma -> Nat
```

som kodar och avkodar naturliga tal som remsor. Skriv dessa funktioner för den kodning som beskrivs i KP, sista stycket, sid 203 där han använder sig av alfabetet $\Sigma = \{\square, 1\}$!

Implementera sedan t ex funktionen `zero` (KP sid 204) och testkör.

Använd gärna din turingmaskinsimulator för att testa dem.

2. KP 6.9–6.11 är inledande övningar i turingmaskinprogrammering. Använd gärna din turingmaskinsimulator för att testa dem. 6.11 är en avsevärt svårare övning.