Litt om invarianter

Stein Krogdahl
Universitetet i Oslo

Magne Haveraaen
Universitetet i Bergen

August 1994*

1 Innledning

Et begrep som ofte brukes i forbindelse med programmering og verifikasjon av progammer er "invariant". Dessverre brukes dette begrepet sjelden eller aldri i lærebøker i datastrukturer og algoritmer. Invariant-begrepet er nært knyttet til teknikken med induksjonsbevis som derimot forekommer oftere. Begrepet anvendes derimot i en del andre kurs ved instituttet, f.eks. i I220. Vi skal derfor forsøke å gi litt bakgrunn for begrepet her, og siden bruke det utover i kurset der det passer. Vi skal imidlertid ikke legge opp til noe formelt forhold til begrepet, slik man har i I220.

Når et program utføres vil programmets data hele tiden forandre seg. Om man bare ser på dataene kan disse forandringene virke svært kaotiske, men det finnes selvfølgelig en idé bak programmet, og det er denne som gir mening og system i alle de forandingene som foregår.

Bruk av invarianter er en måte å få slike program-idéer ned på papiret på, og går rett og slett ut på å angi en betingelse eller et utsagn om variabel-verdiene som "alltid" er sant, eller som i hvertfall er sant hver gang programkontrollen passerer utvalgte steder i programmet.

Disse betingelsene vil derfor representere noe som er fast og uforanderlig i en verden som ellers preges av stadige forandringer, derav navnet invarianter. Vi skal nedenfor se på eksempler på noen slike invarianter.

2 Invarianter er ikke avhengig av matematisk formalisme

Invarianter blir ofte uttrykt i nokså matematisk språk som kan være tungt å lese for folk som ikke er vant til slik formalisme. Denne formalismen har imidlertid lite å gjøre med selve ideen med invarianter, og det er derfor fint om man venner seg til å uttrykke invarianter i vanlig språk der man føler at det er naturlig.

2.1 Tre måter å bruke invarianter på

Når man introduserer invarianter presenteres de gjerne som en teknikk for å kunne bevise at et program har visse egenskaper, og mye av omtalen under kommer også til å dreie seg om denne bruken. Når vi bruker invarianter på denne måten må vi imidlertid være svært nøyaktige og komplette (men merk at dette ikke nødvendigvis henger sammen med det å uttrykke seg i matematisk formalisme). Man kan imidlertid også ha nytte av invarianter selv om man ikke er ute etter å lage fullstendige bevis om programmer. De er da en måte å holde fast på hovedidéen ved programmet mens man lager det, eller når man skal forsøke å sette seg inn i et allerede ferdigskrevet program. Når man bruker invarianter på denne måten kan man bruke den kompletthetsgrad og det presisjonsnivå man finner hensiktsmessig ut fra situasjonen. Invarianter kan

*Revidert august 1999

brukes i betrakninger om alle slags programmer, men i vår sammenheng er det tre situasjoner der de er av spesiell interesse, nemlig i forbindelse med løkker, i forbindelse med rekursive rutiner og i forbindelse med implementasjon av abstrakte datatyper (interface i Java).

3 Løkke-invarianter

Vi skal først se litt på løkke-situasjoner, og vi snakker da gjerne om "løkke-invarianter". Dette er utsagn om variabelverdiene som skal gjelde hver gang utførelsen passerer et nærmere angitt punkt i løkka, og om løkka er en vanlig while-løkke velges dette punktet gjerne umiddelbart før testen på om man skal fortsette utførelsen (også før denne testen gjøres første gang).

Som et enkelt eksempel kan vi se på en liten løkke (med initialisering) som skal summere verdiene i heltallstabellen A fra indeksene 0 til n-1 (deklarasjonen int n og int[] A = int [n] forekommer utenfor vår programbit):

```
int i, sum;
i = 0; sum = 0;

/* Invarianten skal gjelde her */
while (i != n) do
{
    /* Invarianten vil gjelde her */
    sum = sum + A[i];
    i = i+1;
    /* Invarianten skal gjelde her */
};
/* Invarianten og i==n vil gjelde her */
```

Tenker man seg litt om, finner man at den sentrale sammenheng som må gjelde mellom variablene etter hver løkkeomgang er:

```
INVARIANT: sum = A[0] + A[1] + ... + A[i-1]
```

Vi har tidligere reklamert for den dokumentasjonsverdi slike utsagn kan ha (selvfølgelig først og fremst i noe mer kompliserte situasjoner enn dette), men la oss nå se på hvordan slike invarianter kan brukes som den sentrale drivkraften i bevis for at en slik løkke faktisk gjør det den skal. Et slikt bevis får naturlig tre deler:

(a) Vis først at invarianten gjelder umiddelbart før første omgang av løkka utføres (etter initialisering). I eksempelet vårt er både sum og i lik null etter initialiseringen, og vi ser da at invarianten helt trivielt er tilfredstilt (summen av null element er null).

(b) Vis så at dersom løkka ikke er ferdig, så vil innmaten av løkka "bevare invarianten". Dette innbærer å vise at om invarianten gjelder *før* utførelsen og løkka ikke terminerer, så gjelder den også *etterpå*. Dette er som regel den vanskeligste delen å vise, men i eksempelet over er også dette greit nok: Når vi øker i med 1, så må vi også legge det neste elementet i A til sum for at invarianten stadig skal gjelde. Legg her merke til at midt inne i løkka har vi en situasjon der invarianten ikke gjelder, og dette er svært ofte tilfelle. Løkke-innmaten består gjerne av flere setninger som til sammen "gjenetablerer" løkkeinvarianten etter visse forandringer (som forhåpentligvis vil bringe oss nærmere vårt mål). Men så lenge bare noen av disse setningene er utført vil invarianten ofte ikke gjelde.

(c) Vis til slutt at invarianten og betingelsen som gjør at løkka stopper, vil få fram det man faktisk ønsker å oppnå med løkka. I eksempelet er dette svært rett fram. Det vi ønsker å oppnå er:

```
sum = A[0] + A[1] + ... + A[n-1]
```

Men løkka stopper når i==n, og sammen med invarianten følger da dette uten videre.

Merk at dersom man skal gjennomføre slike bevis riktig formelt, så er det gjerne noen ekstra tekniske detaljer som må med i tillegg til det som intuitivt er den sentrale invarianten. For tilfellet over kunne det f.eks. være at i alltid er i området fra 0 til n (og dette vil kreve at n ikke er negativ, noe vi har tatt som en selvfølgelig forutsetning). Vi skal under også se på et liknende bevis for et binærsøke-program.

Selv om eksempelet over gjaldt en while-løkke, så gjelder samme prinsipp for andre løkkevarianter. For en for-løkke av formen

```
/* Invarianten skal gjelde her, for t=a */
for (int t=a,t<=c,t+b)
{
    /* Invarianten vil gjelde her, når løkkevariabelen har verdien t */
    s;
    /* Invarianten skal gjelde her, når løkkevariabelen har verdien t+b */
    };
/* Invarianten vil gjelde her, for t=c+b */
```

vil løkkevariabelen t vanligvis inngå i invarianten og vi må derfor gi den verdi utenom selve løkkekroppen for å kunne gjennomføre resonnementene. Vi kan derfor anta at t har verdien a før løkken begynner og har verdien c+b ved utgangen av løkken[1]. Da antar vi at løkkevariabelen blir oppdatert ved slutten av hver løkkegjennomgang før invarianten skal verifiseres. Siden løkkevariabelen ikke har disse verdiene i selve programmet kan vi ikke bruke disse verdiene av t utenom resonnementet omkring løkken.

Slike bevis kan som nevnt også sees som induksjons-bevis, med induksjon på antall ganger løkka utføres. Invarianten tilsvarer da induksjonshypotesen. Til ettertanke for interesserte nevner vi i denne sammenheng følgende: I likhet med induksjonshypoteser, så vil løkkeinvarianter for mer kompliserte tilfelle ofte være vesentlig mer omfattende enn den delen som gjør at vi kan slutte det ønskede resultat ved terminering. Det er altså ikke bare å "generalisere det ønskede resultat tilbake inn i løkka", slik man kan få inntrykk av fra eksempelet over. Selv om vi kjenner både programbiten og det ønskede resultat, vil det ofte kreves ekte oppfinnsomhet å komme opp med en sterk nok invariant til at det ønskede resultat kan vises. Det kritiske punkt som eventuelt vil avsløre at invarianten ikke er sterk nok, er det å vise at løkke-innmaten bevarer invarianten.

3.1 Termineringsbetingelser for løkker

Dersom vi hadde latt innmaten av løkka være tom, så hadde det vært enda lettere å vise punkt (b) over: Det å ikke gjøre noe i det hele tatt er den sikreste måten å bevare enhver invariant, men det sikrer jo ikke på noen måte at programmet vil levere et fornuftig resultat. Så det vi egentlig har vist med punktene (a), (b) og (c) over er at *dersom* løkka en gang terminerer, *så* vil den gi et fornuftig resultat (dette kalles gjerne "partiell korrekthet"). For å vise at løkka faktisk terminerer må man gi en egen argumentasjon, ved f.eks. å observere at utrykket n-i opplagt vil

[1] Dette er ikke helt sant dersom b ikke går opp i c-a et helt antall ganger. Da vil t==a+n*b for den minste positive verdien av n som gjør at løkken stopper.

minke med 1 i hver omgang, og at når dette uttrykket blir null, så stopper løkka. Dette minner om argumentasjonen for å unngå rekursjonsbrønner i forbindelse med rekursive prosedyrer.

I forhold til FOR-løkker er termineringsresonnementet kurant, da de er garantert å terminere etter et endelig antall steg.

4 Rekursjons-invarianter

Løkker er nær beslektet med rekursive funksjoner, faktisk så nær at et programmeringsspråk i prinsippet bare trenger en av dem for å kunne uttrykke alleslags beregninger. Det er derfor naturlig at reglene for å studere egenskaper ved rekursive prosedyrer er temmelig lik reglene for løkker. For å få en følelse med hvordan dette fungerer vil vi vise at den rekursive prosedyren gen_perm fungerer i henhold til spesifikasjonen angitt i kommentarene etter deklarasjonen[2].

```
void gen_perm(int i){
/* Globale variable: int s, n og int[] p */
/* Forutsetninger: p[s:n] er initiert med data som skal permuteres og
                   "bruk_perm" bruker tabellsegmentet p[s:n] og endrer ikke innholdet i den */
/* Forkrav: s<=i<=n */
/* Resultat: alle permutasjoner av tabellsegmentet p[i:n] genereres for kallet av "bruk_perm",
             og tabellen p settes tilbake til utgangspermutasjonen før retur */
if (i == n) { bruk_perm; } else
    {
    gen_perm(i+1)
    for (int t=i+1,t<=n,t++)
    {
       bytt(i,t);      /* bytter om elementene p[i] og p[t] */
       gen_perm(i+1);
    };
    roter_venstre(i);      /* flytter alle elementene p[i+1], p[i+2], ..., p[n] et hakk mot venstre
                              og elementet p[i] til den ledige posisjonen p[n] */
    };
}
```

Her er s en *mytisk* variabel siden den ikke brukes i selve programteksten, men bare i beskrivelsen av hva prosedyren gjør. Siden spesifikasjonen for gen_perm referer til tabellen p, og denne vil bli endret under utføringen, vil vi trenge en måte å referere til verdien tabellen hadde ved kalltidspunktet i tillegg til den verdien den har for øyeblikket. Dette gjøres ofte ved å sette en indeks $_0$ på variabelen når vi snakker om den initielle verdien. I dette tilfelle vil dermed p_0 betegne verdien av tabellen ved kallet av prosedyren, mens p betegner den verdien tabellen har for øyeblikket. Rekursjonsinvarianten (for alle rekursive prosedyrer) er

- gitt at forkravene til den rekursive prosedyren er oppfylt ved kallet, så vil prosedyren utføre den spesifiserte oppgaven.

Dette kan virke noe banalt siden dette er en selvfølgelighet for enhver prosedyre, enten den er rekursiv eller ikke. Likevel er dette svært viktig å ha klart for seg ved rekursive prosedyrer: *vi kan nemlig bruke det at vi kjenner spesifikasjonen av prosedyren når vi skriver selve prosedyren, og følgelig tenke oss de rekursive kallene som ordinære kall av en hvilkensomhelst prosedyre vi kjenner spesifikasjonen til.* Vi trenger mao. ikke ha oversikt over hvordan rekursjonen oppfører

[2] Selve prosedyren er fra notatet Stein Krogdahl: *Kombinatorisk søking, rekursjon, avskjæring.*

seg under utføring, eller hvordan kallhierarkiet tar seg ut for å skrive eller forstå en rekursiv prosedyre.

Dette viser seg også i hvordan vi viser at en rekursiv prosedyre er korrekt. Strukturen på det som må vises for rekursjonen tilsvarer det vi må vise for løkkeinvarienter, og de tilsvarende punkter er satt opp nedenfor. Rekkefølgen er motsatt av den for løkker for å gjenspeile det som er naturlig ved resonnement om rekursive prosedyrer.

(a) Vis først at de innerste kallene av prosedyren bevarer invarianten og oppfyller prosedyrens spesifikasjon. De innerste kallene er kall der parametrene er slik at prosedyren ikke kaller seg selv fler ganger. I eksempelet inntreffer dette når i==n, og siden bruk_perm ikke endrer p er den siste delen av resultatbeskrivelsen oppfyllt. Den første delen blir også oppfylt, fordi bruk_perm blir kalt for alle mulige permutasjoner av det ene elementet p_0[n].

(b) Vis så at dersom rekursjonen ikke er ferdig, så vil innmaten av prosedyrekroppen "bevare invarianten". Dette innbærer at for alle kombinasjoner av parameterverdier som fører til nye rekursive kall, så må rutinen bevare rekursjonsinvarianten, dvs. oppfylle sin egen spesifikasjon. Dette er som regel den vanskeligste delen å vise, men når vi skal vise dette, kan vi anta at alle indre prosedyrekall oppfyller spesifikasjonen.

I eksempelet vårt inntreffer dette for ethvert kall med s<=i<n, dvs. resten av de lovlige kallene. I dette tilfellet kaller vi gen_perm(i+1) gjentatte ganger. Det første kallet skjer uten at vi har endret p, og vil etter antakelsen vår føre til at bruk_perm utføres for alle permutasjoner av tabellsegmentet p[i+1:n] med p[i] i første posisjon. Resten av kallene utføres inne i en løkke, og vi må da vise at denne løkken oppfører seg riktig. Dette krever en løkkeinvariant som det som alltid kreves en del innsikt for å kunne sette. Som løkkeinvariant kan vi bruke

- tabellsegmentet p[i:t-1] endres til å inneholde p[i]=p_0[t-1], p[i+1]=p_0[i], ..., p[t-1]=p_0[t-2], mens resten av tabellen p er urørt, og
- bruk_perm er kalt for de permutasjoner av p_0[i:n] som har elementene p_0[i], p_0[i+1], ..., p_0[t-1] i posisjon p[i].

Det er viktig at invarianten i tillegg til endringen av p fanger opp kravet med kallene av bruk_perm etter som løkken skrider fremover.

Før starten av løkken kan vi anta at t==i+1, og vi går inn i den med de opprinnelige tabellverdiene, dvs. at løkkeinvarianten er tilfredsstilt. For å se at løkke-kroppen bevarer løkkeinvarianten legg merke til at ombyttingen setter p_0[t-1], som var i p[i] inn i p[t], og elementet p_0[t] inn i p[i]. Kallet av gen_perm(i+1) endrer ikke dette, men utfører kallet av bruk_perm for permutasjonene av p[i+1:n] med p_0[t] i posisjon p[i]. Dermed er løkkeinvarianten etablert for neste verdi av t. FOR-løkken slutter når t==n, og da sier invarianten at alle elementene er forskjøvet en posisjon mot høyre, unntatt elementet p_0[n] som er i p[i]. Dessuten er bruk_perm utført for alle permutasjonene av p[i+1:n] med elementene p_0[i], p_0[i+1], ..., p_0[n] i posisjon p[i], hver gang med resten av elementene fra p_0[i:n] i segmentet p[i+1:n]. Til sammen betyr det siste at bruk_perm er utført for alle permutasjoner av p[i:n]. Den første delen av invarianten for løkken betyr at venstrerotasjonen til sist i prosedyren gen_perm bringer alle elementene tilbake i utgangsstilling. Følgelig oppfyller gen_perm spesifikasjonen også i dette tilfellet.

(c) Vis til slutt at rekursjonsinvarianten gjelder ved de ytre kallene av prosedyren. Dette hører ikke egentlig til det å sjekke at prosedyren er programmert rett, men til det at den brukes rett, akkurat som kravet er til enhver prosedyre.

Akkurat som for resonnement omkring løkker kan argumentasjonen utføres mer eller mindre formelt. Fullstendig formelle bevis krever gjerne en god del mer presisering enn de uformelle bevisene, men et uformelt bevis kan ha minst like god effekt for å luke bort unødvendige feil og få de siste detaljer i et program på rett.

Bevis av rekursive prosedyrer har også sterke likhetstrekk med induksjonsbevis. Punkt (a) svarer da til å etablere basistilfellene, mens i (b) tar vi for oss de mer kompliserte tilfellene basert på at induksjonshypotesen (spesifikasjonen av den rekursive prosedyren) er korrekt. Siden vi bare unntaksvis vil ha rekursive prosedyrer på den enkle formen med en indeksvariabel som endrer seg i steg på 1 (som i tilfellet med gen_perm), så kalles denne formen for induksjon *strukturell induksjon* (vi tar for oss flere basistilfelle og alle måter disse kan kombineres på), i motsetning til ren induksjon over naturlige tall som er vanlig fra begynnende matematikk[3].

4.1 Termineringsbetingelser for rekursive prosedyrer

Det som ofte er vanskelig å forstå med rekursive prosedyrer er hvordan det å skrive en prosedyre som kaller seg selv virkelig kan få utført noe som fører til en løsning. Som for løkker gjelder at betingelsene som viser invariantens bevaring ikke er sterke nok til å garantere dette. F.eks. vil følgende prosedyre tilfredsstille punktene over:

```
void r()                          /* DENNE METODEN FUNGERER IKKE */
/* Forkrav: data initialisert */
/* Resultat: alle problem er løst */
{ r() };
```

Problemet her er opplagt: det skjer ingenting her utenom det rekursive kallet, noe som fører til en uendelig dyp rekursjonsbrønn. Vi må altså ha setninger i prosedyren som garanterer at det skjer "noe" før hvert rekursivt kall.

Å vise dette kan gjøres med noe som teknisk kalles velordninger: dataene som behandles gis en ordning, og vi passer på at hvert kall skjer "lenger nede" i ordningen. Ordningen må være slik at det bare finnes et rimelig antall steg før den rekursive prosedyren går inn i den ikke-rekursive delen. I vårt eksempel betyr dette at vi ser på parameteren i til prosedyren og legger merke til at den økes med 1 for hvert rekursivt kall til rekursjonen stopper når i==n. Følgelig nærmer hvert rekursivt kall seg en avslutning, dersom de andre setningene i prosedyren fullfører. Det forutsetter at løkker etc. i kroppen til prosedyren terminerer, her en FOR-løkke som er garantert å terminere. Dermed bør vi føle oss temmelig sikre på at gen_perm faktisk fullfører jobben sin.

5 Data-invarianter

Den andre hovedtypen av invarianter er betingelser som vi vil skal gjelde for den datastrukturen som representerer en instans av en abstrakt datatype (eller for en annen vel avgrenset datastruktur i et program). Slike invarianter kalles "data-invarianter" eller "representasjons-invarianter", og vil gjelde "hele tiden" etter at variablene i instansen er initialisert, bortsett fra mens en operasjon på den abstrakte datatypen er under utførelse. Prosedyrene som implementerer operasjonene for den abstrakte datatypen spiller her samme rolle som innmaten i løkka for løkkeinvarianter: invarianten kan godt brytes mens prosedyren er under utførelse (og det må den jo som regel, siden alt ikke kan forandres på en gang), men den må gjenopprettes før prosedyren terminerer.

Når vi skal overbevise oss om at de forskjellige operasjonene i den abstrakte datatypen virker som de skal kan vi dermed anta at invarianten gjelder hver gang en operasjon starter, og i tillegg

[3]Det er mer stoff om induksjonsbevis og varianter av den i kursene I127 og I234.

til å vise at operasjonen leverer riktig resultet etc., må vi også sørge for at invarianten fremdeles gjelder ved prosedyrens avslutning.

Som et eksempel kan man her tenke seg en abstrakt datatype som representerer en mengde av tall, der man har operasjoner for å sette inn tall, ta ut tall, og sjekke om et gitt tall er med i mengden. En mengde er bare en samling $\{x_1, x_2, \ldots, x_n\}$ av elementer x_i, der det er implisitt at hver verdi forekommer bare en gang, selv om den er listet opp flere ganger. Det kan lett tenkes flere implementasjoner av en slik datastruktur. En av de mer rett frem er å sette av en tabell med nok plass, f.eks. A[L], og så lagre elementene der. Hvordan elementene skal håndteres i tabellen kan gjøres på flere måter, og vi skal illustrere 2 varianter som skiller seg ved datainvarianten brukt:

- Vi kan som et enkelt alternativ velge å lagre elementene hulter til bulter i de første n posisjoner i tabellen, og heller ikke bry oss om en verdi blir gjentatt flere ganger. I så fall kan det tenkes n er større enn det virkelige antall elementer i mengden. Denne invarianten kan formuleres som

  - 0 <= n <= L, og
  - bare elementene A[i], slik at 0<=i<n, teller med i datastrukturen.

  Den siste delen av datainvarianten forteller oss at innholdet i tabellsegmentet A[n:L-1] er uinteressant, selv om det ikke forteller oss hvordan vi skal oppfatte verdiene i segmentet A[0:n-1] utover at de skal representere mengden. Denne invarianten fører til at innsetting blir svært lett (bare føy inn den nye verdien bakerst), mens både det å ta ut (let gjennom hele tabellen og fjern alle forekomster av verdien ved å kompakte tabellen) og å sjekke om et gitt tall var med (søk sekvensielt gjennom tabellen til vi har funnet en forekomst av verdien, eller vi er kommet til slutten av tabellen) vil bli tunge.

- En variasjon over dette er å ha dataene sortert i tabellsegmentet. Datainvarianten kan da skrives:

  - 0 <= n <= L,
  - bare elementene A[i], slik at 0<=i<n, teller med i datastrukturen, og
  - A[0] < A[1] < ... < A[n-1]

  Her forteller det siste leddet i invarianten at alle dataene i tabellen er forskjellige, og at de er lagret i stigende rekkefølge.

  Denne sammenhengen kan dermed operasjonene anta gjelder når de starter, men de står også ansvarlig for at den gjelder når de er ferdige. For sjekke-operasjonen er dette bare en fordel siden den ikke endrer dataene, men nå kan gjøre rask binærsøking. For de to andre operasjonen er denne invarianten mer av en belastning. De kan riktignok raskt finne hvor et element skal inn eller ut, men må ofte gjøre en større forflytning for å få invarianten til å gjelde etter innsettingen eller uttaingen.

Og denne situasjonen er typisk: Når man forandrer datainvarianten vil noen operasjoner vinne på det og andre tape, og hva som er det gode valg vil avhenge av situasjonen. Om vi skulle finne på å lage en operasjon som forteller hvor mange element som er i mengden, så vil det bli lett å implementere med den siste invarianten, da den forteller oss at alle elementene er forskjellige, slik at svaret blir n. Med den første invarianten blir dette mye vanskeligere, siden vi da måtte trekke fra for dobbelt-forekomster.

Slike data-invarianter vil med fordel kunne angis for nesten enhver avgrenset datarepresentasjon i et program, og de vil altså representere de betingelser vi har lagt på dataene for at de skal ha mening, og kunne behandles på den måten vi har planlaget. Og derfor: man bør alltid angi som kommentar til hver avgrenset datastruktur i et program de invarianter som knytter seg til denne datastrukturen. Ofte kan det være lurt å formulere denne som en egen metode `boolean DI()`. `DI()` kan da brukes til å sjekke om dataene i datastrukturen er konsistent med forutsetningene. Når vi utvikler den abstrakte datatypen vil vi bruke `DI()` til å sjekke om hver av operasjonene gjenetablerer datainvarianten ved retur. En vanlig feil, hvis vi ikke dokumenterer datainvarianten, er at vi f.eks. tenker på den andre invarianten i eksemplet over når vi skriver sjekke-prosedyren, men glemmer sorteringskravet når vi skriver innsettings-prosedyren. I tillegg til å sjekke selve modulen, kan vi også bruke metoden `DI()` til å sjekke om noe er gått galt om vi mistenker andre deler av programmet for å manipulere direkte med dataene i datastrukturen. Dette kan skje om ikke typesystemet i språket garanterer god nok beskyttelse av abstraksjonene våre.

I forbindelse med de datastrukturer vi ser på i kurset (så som forskjellige typer tabeller, trær og grafer) vil det stadig knytte seg forskjellige typer invarianter, og de vil ofte være vesentlig mer subtile enn den vi så på over.

5.1 Abstraksjonsfunksjoner

Et annet begrep som dukker opp i forbindelse med datastrukturer er abstraksjonsfunksjoner. Abstraksjonsfunksjoner forutsetter at vi har en spesifikasjon av datastrukturen vi holder på å implementere. Abstraksjonsfunksjonen representerer da den sammenheng vi har valgt å ha mellom den konkrete datastrukturen og de begrep som brukes i spesifikasjonen. Den angir altså hva en gitt datastruktur (som oppfyller invarianten) skal "bety". Abstraksjonsfunksjon "oversetter" fra den konkrete datastrukturen til de begrep som brukes i spesifikasjonen, og gir dermed den informasjonen som manglet i eksemplet over, nemlig hvordan dataene skal fortolkes som en mengde.

I eksemplet vil abstraksjonsfunksjonen uttrykke at elementene i tabellsegmentet `A[0:n-1]` utgjør mengden, dvs. samlingen {`A[0]`, `A[1]`, ..., `A[n-1]`}. Vi overser dermed indeksene, rekkefølgen elementene har i tabellen og elementene i segmentet `A[n:L-1]` ved oversettelsen. Denne abstraksjonsfunksjonen kan vi i dette spesielle tilfellet bruke for begge representasjonene av mengden i eksemplet, men dette er ikke vanlig. Abstraksjonsfunksjonen trenger ikke være meningsfull dersom datainvarianten ikke er oppfyllt, f.eks. ved at `n` ligger utenfor det lovlige intervallet, eller at en av indeksposisjonene i `A[0:n-1]` er tomme.

Abstraksjonsfunksjonen (`AF`) spiller en viktig rolle når det gjelder å vise at en implementasjon er korrekt. Når vi har gitt en spesifikasjon på det abstrakte plan og en implementasjon på det konkrete plan, må vi sjekke at det ønskede samsvar er til stede: de endringer koden utfører på dataene må svare til de endringer som er beskrevet i spesifikasjonen. For hver prosedyre må vi derfor vise at diagrammet under kommuterer (dvs. at vi får samme virkning om vi går ned først og så bort, eller bort først og så ned):

$$
\begin{tikzcd}
DS_0|DI \arrow[r, "AF"] \arrow[d, "K"'] & AS_0 \arrow[d, "SP"] \\
DS_1|DI \arrow[r, "AF"] & AS_1
\end{tikzcd}
$$

Her betyr DS|DI innholdet i den konkrete datastrukturen DS, med datainvarianten DI() oppfyllt, og AS representerer de abstrakte verdiene fra spesifikasjonen. Subskript $_0$ betegner verdiene før prosedyren kalles, mens subskript $_1$ betegner verdien ved retur. K er koden fra implementasjonen, AF er abstraksjonsfunksjonen, og SP er virkningen av prosedyren i henhold til spesifikasjonen.

I andre del av vårt eksempel over vil $DS_0|DI$ være tabellen A[L] med de sorterte elementene i segmentet A[0:n-1], og AS_0, som vi får ved å følge pilen bort, vil være mengden av elementene fra dette segmentet, slik AF foreskriver. Studerer vi nå innsett-funksjonen finner vi at resultatet av dennes spesifikasjon SP (pil ned på høyre siden i diagrammet) blir en mengde (AS_1) med det nye elementet i tillegg til de gamle elementene i mengden. Vi må nå sjekke at implementasjonen er korrekt ved å følge pilene ned på venstre siden og så bort, og se at vi får den samme verdien. Koden K står for implementasjonen av innsett-prosedyren. Implementasjonen foreskriver at det nye elementet blir satt inn på sortert plass i det sorterte tabellsegmentet, og at n økes med 1. Dette er den nye konkrete dataverdien $DS_1|DI$ som vi får ved å følge pilen ned på venstre siden i diagrammet. Den nye abstrakte verdien AS_1, som vi får ved å gå bortover i diagrammet, blir den nye mengden. Den finnes ved å samle alle elementene i den nye tabellen, dvs. den gamle tabellen med det nye element sortert inn, i en mengde. Dermed blir svaret likt enten vi går først bort og så ned, eller først ned og så bort, og vi har dermed vist at implementasjonen er rett.

Denne prosessen må gjentas for slette- og oppslagsfunksjonene også. I det siste tilfellet blir arbeidet noe annerledes siden vi ikke endrer på tabellen eller mengden. Da får vi inndataene (tabellen og mengden hhv.) øverst i diagrammet, og resultatet av funksjonen, svaret sant eller usant, nederst i diagrammet. Abstraksjonsfunksjonen for de boolske verdier er svært triviell (det blir jo en identitet), men det er likevel viktig at implementasjonen valideres mot spesifikasjonen, dvs. å vise at svaret blir likt på begge sider.

Abstraksjonsfunksjonen kan også være nyttig ved uttesting og feilsøking i et program. Ikke ved at den direkte kan programmeres for å teste verdier, men ved at vi kan lage utskriftsprosedyrer som gjengir "det abstrakte innholdet" av en datastruktur slik abstraksjonsfunksjonen beskriver den. Dersom vi så har mistanke om at programmet legger feil data inn i mengden i eksemplet over, kan vi skrive ut innholdet av mengden for å sjekke dette på utvalgte steder i programmet. Det som da er viktig er at utskriften lister dataene slik vi tenker på dem ved bruk, nemlig som om det var en mengde og ikke en tabell. Spesielt er dette viktig dersom den abstrakte datatypen har en enkel beskrivelse, mens implementasjonen er svært avansert.

6 Invarianter kan prege programutformingen

Slik begrepet "invariant" er framstilt over, representerer det noe vi kan knytte til en programtekst som er ferdig skrevet. Ofte vil det imidlertid være omvendt. Når man skal gi et program struktur, kan det være et antall invarianter som mer eller mindre eksplisitt er med å prege den struktur man velger. Dette vil igjen si at om man vil ha god struktur på sitt program, så må man ha tak i de "dypeste" invariantene i den problemstillingen man arbeider med, akkurat som den elegante løsning av en fysikkoppgave ofte er å gjøre en betraktning omkring energiens bevarelse: energi forsvinner ikke, den går bare over i stadig nye former. Begrepet "energi" er altså en invariant i naturen.

6.1 Invarianter brukt i binærsøking

Vi skal her se litt på hvordan invarianter kan hjelpe oss til å få kontroll med detaljene i en algoritme, og til å kunne føle oss tryggere på at programmet gjør det det skal. Vi ser på et (Java)program for binærsøking. Som underliggende forutsetning antar vi at heltallsverdiene i tabellen A fra indeks 0 til n-1 er sortert, og at vi skal undersøke om verdien x finnes blant disse. Programmet er som følger:

```
{
    int low, mid, high;
    boolean found;
    low = 1; high = n; found = false;
    while (low <= high && ! found)
    {
        mid = (low + high) / 2;
        if (A[mid] < x) { low = mid+1; }
        else if (A[mid] > x) { high = mid-1; }
        else  found = true /* Vi vet A[mid] == x */ ;
    };
    answer = found ? mid : -1;
};
```

Vi ser at programmet leverer 0 i answer dersom verdien x ikke finnes, ellers leveres indeksen i A til elementet der verdien ligger. Vi skal konsentrere oss om while-løkka i programmet, og vi påstår at ideen med denne kan utrykkes ved følgende invariant:

- Dersom found = true så er 1 <= mid <= n og A[mid] = x
- For alle i = 1, 2, ... , low-1, så er A[i] < x
- For alle j = high+1, high+2, ... , n, så er A[j] > x
- low <= high+1 og 1 <= low <= n+1 og 0 <= high <= n

Den siste linja er typisk slike tekniske detaljer som kan være viktige i mer presise bevis.

Vi har to hovedpoeng: For det første kan man, før man starter å programmere, bestemme seg for at nettopp denne invarianten skal gjelde for programmet eller programbiten, og dette vil kunne hjelpe en til å få initialisering, flytting etc. av indekser riktig på plass slik at man f.eks. unngår de typiske pluss-minus-en feil som lett oppstår[4]. For å få litt føling med hvordan dette kan

[4]Boken *The science of Programming* (Springer, 1981) av David A. Gries bruker dette som en systematisk teknikk for programutvikling

spille seg ut kan man løse oppgave 2 under, som består i å lage en annen variant av binærsøking ut fra en noe modifisert invariant.

Det andre poenget er at vi, når vi har programmet på plass og vil ettersjekke at det virker, kan bruke den tre-punkts-metoden som er angitt i avsnittet om løkke-invarianter over. Ser vi først på punkt (a), så må vi altså sjekke at etter initialiseringen vil invarianten gjelde. Om vi bare forutsetter at n ikke er negativ, så vil dette gjelde uten videre siden found er false og siden de mengdene av verdier for i eller j det kreves noe om, begge er tomme.

Punkt (b) er så å vise at én omgang i løkka vil bevare invarianten. Vi skal ikke gå i detaljer her, men bare henvise til at om man ikke er ferdig så vet man at low<=high, og ut fra måten mid beregnes på så vet vi også low<=mid<=high. Programmet skiller så mellom de tre tilfellene A[mid]<x, A[mid]>x og, A[mid]=x, og det å sjekke at invarianten bevares i hver av disse tilfellene, overlates til leseren.

Endelig må vi sjekke punkt (c), altså at vi ut fra invarianten og det at løkka terminerer kan slutte at det som er hensikten med løkka er oppnådd. At løkka terminerer innebærer enten at found==true eller at high<low. I første tilfelle ser vi uten videre (av første punkt i invarianten) at verdien x ligger i indeksen mid. I det andre tilfellet følger det av de to midterste punktene i invarianten at alle verdier i A enten må være mindre enn eller større enn x, og at verdien derfor ikke kan finnes i A.

Den siste setningen i programmet henter ut svaret nettopp ut fra dette, og programmet skulle derfor være (partielt) korrekt. At det også vil terminere følger av at i hver omgang må enten high-low minke eller found må bli true.

7 Noen oppgaver

7.1 Oppgave 1

Gitt en variabel V, med datainvariant DI(), i et program P. For å bevise at V alltid oppfyller datainvarianten DI(), er følgende forutsetninger for et induksjonsbevis satt opp:

- **induksjonsbasis**: V oppfyller datainvarianten etter initiering. Dette svarer til 0'te oppdatering
- **induksjonshypotese**: Hvis V oppfyller datainvarianten etter n oppdateringer, så vil V oppfylle invarianten etter oppdatering $n + 1$.

Hva må til for at disse forutsetningene skal være oppfyllt i P?

Vil det bli lettere å gjennomføre beviset dersom vi har samlet all kode som kan referere direkte til datastrukturen for V i en egen modul?

Må vi gjennomføre beviset for hvert program P, eller kan vi klare beviset en gang for alle?

Argumenter ut fra dette at datainnkapsling (begrensning av synligheten til en datastrukturdeklarasjon) og sterk typing (variable kan bare brukes slik datastrukturen gir adgang til) er gode egenskaper i et programmeringsspråk.

7.2 Oppgave 2

I den binærsøkealgoritmen som er diskutert i teksten over blir det gjort to tester (for å skille mellom tre tilfeller) i hvert gjennomløp. I hvertfall i starten er det tredje tilfellet (det at A[mid]==x) lite sannsynlig, og den siste testen er derfor som oftest unødvendig. Oppgaven her er å lage en variant av binærsøking som i hvert gjennomløp bare skiller mellom de to tilfellene A[mid]<=x og

A[mid]>x, og som derved utsetter avgjørelsen på om x fantes eller ikke helt til slutt. Den sentrale delen av invarianten for denne kan være:

- For alle i = 1, 2, ... , low-1, så er A[i] <= x
- For alle j = high+1, high+2, ... , n, så er A[j] > x

7.3 Oppgave 3

En *halerekursiv* prosedyre er en rekursiv prosedyre med bare et rekursivt kall, og der dette kallet kommer helt til slutt i prosedyrekroppen (i halen). Prosedyren gjenta til venstre under er halerekursiv. Til høyre står en løkke, med de samme betingelser, setninger og invarianter som for prosedyren.

```
void gjenta()
/* Forkrav: ⟨invariant⟩ skal være oppylt */
/* Resultat: ⟨invariant⟩ og ⟨sluttbetingelse⟩ er oppfylt */
{
/* ⟨invariant⟩ gjelder */
if (⟨sluttbetingelse⟩)
    { /* ⟨invariant⟩ og ⟨sluttbetingelse⟩ er oppfylt */ }
else {
    /* ⟨invariant⟩ er oppfylt */
    ⟨løkkekroppen⟩;
    /* ⟨invariant⟩ er oppfylt */
    gjenta();
    /* ⟨invariant⟩ og ⟨sluttbetingelse⟩ er oppfylt */
    };
/* ⟨invariant⟩ og ⟨sluttbetingelse⟩ er oppfylt */
};
```

```
/* ⟨invariant⟩ er oppfylt */
while (⟨sluttbetingelse⟩)
{     /* ⟨invariant⟩ er oppfylt */
    ⟨løkkekroppen⟩;
    /* ⟨invariant⟩ er oppfylt */
    };
/* ⟨invariant⟩ og ⟨sluttbetingelse⟩ er oppfylt */
```

Vis at den rekursive prosedyren og løkken begge gjør det samme, dvs. at dersom de startes for de samme verdiene vil de levere de samme resultatene.

Det er i prinsippet ingen forskjell på uttrykkskraften til rekursive prosedyrer og løkker, og mange programmeringsspråk klarer seg med en av dem. Vanligvis vil koden som lages ved oversetting av rekursive prosedyrer kreve mer administrasjon enn den som lages for løkker, slik at det blir tyngre å utføre et program basert på rekursive prosedyrer enn et tilsvarende program skrevet med løkker. Nå er ikke denne forskjellen større enn at den ikke bør tillegges vekt med mindre programmet er svært tidskritisk. Enkelte kompilatorer klarer også å oversette halerekursive funksjoner til løkker, slik at mange programmerere ikke "ser" forskjell på dem. Det viktige er å bli kjent med begge uttrykksformer, og så bruke den som er mest hensiktismessig ut fra problemet som skal løses.