Tensorfelt, fra Spesifikasjon til Implementasjon.

Victor Madsen

Universitetet i Bergen,
Institutt for Informatikk,
studieretning Databehandling

10. juni 1993

Til min kommende kone ...

Dette arbeidet har vært et samarbeid mellom forskjellige disipliner innen databehandling. Hans Munthe-Kaas har støttet meg med den nødvendige matematiske teori i oppgaven. Det var han som først så et behov for et arbeide innen dette området, og det er han som har skissert opp målet for oppgaven. Den teoretiske og praktiske erfaring til Hans har vært til stor nytte.

Magne Haveraaen fant at algebraisk spesifikasjon ville være en nyttig metode for å formalisere våre matematiske strukturer. Han skrev og et tidlig forslag til algebraisk spesifikasjon av noen av de grunnleggende strukturene. Det var og han som først foreslo at vi skulle velge en 'tensor over felt' løsning i stedet for en løsning med 'felt over tensor'. Dette hadde avgjørende betydning for dette arbeidet.

Jeg har fulgt opp de forslag til metodevalg som Hans og Magne har foreslått. Jeg har strukturert den matematiske delen og definert og formalisert det vi senere skal referere til som *deriverbare strukturer.* Videre har jeg definert *Lie derivasjon* og *partiell derivasjon* på disse strukturene. Jeg har og utviklet og implementert de algoritmene og metodene som er nevnt i oppgaven.

Jeg vil rette en stor takk mine veiledere Hans og Magne. Begge har alltid vært velvillige og tolerante, særlig når en tenker på mine stadige (ikke avtalte) besøk på deres kontor. Deres støtte har vært langt større enn hva en kan forvente av dem, og deres faglige kommentarer har hatt alt å si for min oppgave. Jeg vil særlig takke Hans som ga meg muligheten til å reise på konferanse i USA. Reisen betydde mye for meg både faglig og privat.

Victor Madsen

Contents

Chapter 1

Innledning

Differensiallikninger er av grunnleggende betydning innenfor fysikk og matematikk siden svært mange fysiske lover og sammenhenger kan formuleres matematisk som slike likninger. Vi kan si at differensiallikninger er likninger som inneholder den deriverte av en eller flere ukjente funksjoner. Utfordringen ligger da i å finne den ukjente funksjonen. Partielle differensiallikninger er likninger som inneholder den deriverte av funksjoner over to eller flere uavhengige variabler. De uavhengige variablene representerer gjerne tiden og en eller flere romkoordinater. Et enkelt eksempel på en partiell differensiallikning (PDL), er:

$$\frac{d^2}{dt^2}(u(x,t)) = c^2(x) * \frac{d^2}{dx^2}(u(x,t))$$

Det finnes mange PDL'er som ikke kan beregnes analytisk, og man må da benytte seg av datamaskiner for å finne numeriske løsninger av likningene. Dette er ikke en trivell jobb, og mange av verdens kraftigste datamaskiner bruker mye av sin kapasitet for å løse slike likninger.

Vanligvis vil et program som simulerer PDL'er være implementert svært eksplisitt med diskretiseringsmetoder av det underliggende rom-tid kontinium. Dette får konsekvenser for hele programmet, da implementasjonsstrategi og metodevalg vil være representert i mesteparten av kildekoden. Dette er en svakhet, siden en da binder seg sterkt opp mot en løsningsmetode, og det vil kreve mye programmering for å endre f.eks. diskretiseringsmetoden. Dette gjør og at gjenbruk av deler av koden blir vanskelig siden kildekoden er spesialisert opp mot ett spesiellt problem. Det fører og til at feilsøking, velikehold og utvidelse av kildekode kan være et stort problem.

Innenfor numerisk databehandling er det stor motstand mot dagens høynivå programmeringsspråk. Den viktigste grunnen til dette er frykten for tap av effektivitet. Det er en vanlig antagelse at et lavnivå språk som FORTRAN vil generere mer effektiv kode enn f.eks. PASCAL og C++. Dette er ikke et helt uvesentlig poeng, men det avhenger av hvordan man måler effektivitet. Hvis man ser på de totale kostnader for et program, må man ikke glemme utviklingstiden. Det er allment kjent at man i et høynivå språk ofte kan utvikle (og feilrette !) svært komplekse program mye raskere enn i et lavnivå språk. Noe som ofte er like viktig, er vedlikeholdskostnadene. Et annet poeng er at kostnadene forbundet med maskinvare synker raskt mens lønnskostnadene er forholdsvis uendret. Dette betyr at det blir mer og mer interessant å redusere kostnadene forbundet med utvikling og vedlikehold.

Vi ønsker å benytte metoder og prinsipper fra programmeringsteknologi for å skrive bedre kode for løsning av partielle differensiallikninger. Med bedre kode mener vi her kode som:

- er lettere å **vedlikeholde**
- støtter **gjenbruk**
- er **flyttbar** mellom forskjellige maskinarkitekturer
- støtter **raskere utvikling** av nye program
- er **effektive** med hensyn på kjøretid

En grunnleggende stuktur er *tensorer*, og det er kjent at de fleste PDL'er kan formuleres som en koordinatfri likning mellom tensorer uavhengig et underliggende koordinatsystem. At likningene er koordinatfrie viser seg å være helt essensielt for oss, siden vi da klarer å frigjøre oss fra nødvendigheten av å vite hvilke diskretiseringsvalg som er gjort. Beregninger på tensorfelter har en mengde innebygd parallellitet i algoritmene, og egner seg derfor svært godt for parallelle maskiner. Det er ikke triviellt å implementere beregninger på tensorfelter siden det lett blir mange indekser å holde kontroll over ved de tradisjonelle programmeringsteknikkene. Vi får først og fremst en mengde indeksmanipulasjoner ved beregninger av tensorlikninger, og i tillegg får vi enda flere indekser når vi skal representere *felter* av tensorer. En implementasjon av beregninger på en parallell maskin vil ofte være være svært avhengig av den aktuelle maskinarkitektur, og en må ofte forvente en fullstendig omskriving av koden hvis vi skal flytte den over til en ny arkitektur. Utgangspunktet for dette prosjektet var at vi skulle implementere et bibliotek av funksjoner som behandlet tensorer . Disse funksjonene skulle ha et enkelt grensesnitt der vi behandler tensorer som objekter i en klasse. Vi ville da innkapsle all manipulasjon av indekser inne i noen generelle tensorfunksjoner. Vi skulle videre implementere en feltversjon av tensorklassen vi utviklet. Denne feltimplementasjonen av tensorer skulle inneholde rutiner som utførte tensorberegninger på et helt felt av tensorer. Denne tensorfeltklassen ville ikke være avhengig av noen spesiell maskinarkitektur, og ville derfor kunne implementeres på en rekke forskjellige maskiner. Ved å ha et fast grensesnitt mot samtlige implementasjoner av tensorfelter vil vi oppnå at et program som benytter seg av denne klassen uten videre vil kunne rekompileres på alle parallelle maskiner som har en implementasjon av klassen. Vi vil da kunne implementere bedre programmer (med hensyn på punktene over) enn det som i dag blir gjort.

Oppgaven virket forholdsvis enkel. Det var en del detaljer av implementasjonsteknisk art, men vi visste hva vi ønsket og vi trodde vi visste hvordan det skulle gjøres. Dette virket derfor som en svært praktisk oppgave. Vi fikk veldig raskt problemer når vi skulle realisere vår innfallsvinkel til problemet. Det var flere grunner til dette:

- Hva er en tensor ?
- Hva er et felt av tensorer ?
- Hva er en derivasjonsoperator ?
- Hvilke derivasjonsoperatorer skulle vi implementere ?

Vi skulle implementere et sett av generelle funksjoner som utfører operasjoner på tensorer. Det er ikke opplagt hva en tensor er, og det finnes mange forskjellige definisjoner av dem. Vi måtte derfor studere nærmere en del matematisk litteratur for å finne hvilke operasjoner som er naturlig å ta med i en implementasjon. Noe som viste seg å være enda vanskeligere er å finne ut hva et felt er. Vi tenkte i utgangspunktet på et felt som et slags gitter, men dette var ikke noen god innfallsvinkel. Grunnen er at vi da ikke kunne benytte en del andre diskretiseringsteknikker som f.eks. *endelig element* metoden. Hvilke operasjoner er det vi skal ha med i en tensorfeltklasse som vi ikke har med i en tensorklasse? Det har ikke noen mening å snakke om derivasjoner hvis vi ikke har noen felter å derivere over, så det virket som det var en sammenheng mellom derivasjonsoperatorer og felter. Vi måtte derfor studere denne sammenhengen nærmere. Denne oppgaven endret seg derfor raskt fra en rent praktisk oppgave til å inneholde et sterkt teoretisk element. Vi endret derfor etter hvert målsettingen for oppgaven til bare å utvikle tensorklassene, og ikke være så opptatt av å få en parallell implementasjon med en gang. Dette viste seg etter hvert å være et mer realistisk mål for oppgaven.

Vår innfallsvinkel for å løse problemene var å skille klart mellom *hva* en ønsker at operasjonene skal gjøre og *hvordan* det skal bli gjort. Innenfor den matematiske disiplinen *global analyse* har dette lenge vært et hovedpoeng. Global analyse legger vekt på finne stukturer og sammenhenger, og dette vil vi benytte for å finne mye av den nødvendige matematiske teori. Innenfor global analyse ([1]) beskriver man et tensorfelt som en *multilineær avbildning* over *tangentbunken* til en *manifold.* En manifold er en ganske kompleks struktur, og vi har i denne oppgaven begrenset oss til strukturer isomorfe med *parallelliserbare mangfoldigheter.* Denne begrensningen inntreffer først i kapittel 6, der vi begynner å konkretisere definisjonene vi formaliserer til implementerbare modeller.

Vi benytter *algebraisk spesifikasjon* for å systematisere og formalisere de matematiske strukturene som er gitt i global analyse. Dette gjør at vi klarer å formalisere strukturer og relasjoner, og vi finner en nær sammenheng mellom disse og abstrakte datatyper (ADT'er). Algebraisk spesifikasjon hjelper oss å finne hvilke ADT'er vi trenger, og hvilke operasjoner de skal ha. Vi sier ikke noe om hvordan operasjonene skal utføres, vi bare spesifiserer hvilke egenskaper de skal ha. Det viser seg at denne innfallsvinkelen er svært nyttig, da vi automatisk oppnår en høy grad av modularisering. Dette er svært viktig, siden vi da kan implementere ADT'er som vi senere kan bruke som 'byggeklosser' for å sette sammen ferdig kjørbare program. Fordelene med denne fremgangsmåten er mange:

- Vi kan gjemme implementasjonsvalg inne i små moduler.
- Det blir lettere å vedlikeholde applikasjoner.
- Gjenbruk av kode blir redusert til gjenbruk av moduler.

Summen av alt dette er at vi vil oppnå høyere kvalitet på vår kode samtidig som vi reduserer kostnadene forbundet ved utvikling og vedlikehold. Det vil og vise seg at de modulene vi ender opp med vil være lette å parallellisere, siden all kode som trenger å parallelliseres kan kapsles inn i en liten modul med et enkelt og naturlig grensesnitt.

Vi implementer til slutt alle strukturene våre som klasser i C++, og utvikler et program som benytter disse klassene. Dette programmet er et eksempel på hvordan vi kan utføre *seismisk modellering* ved hjelp av likninger på tensorform, og vi ser her at det er en nær sammenheng mellom tensorlikningene og et kjørbart program som benytter klassene vi har utviklet.

Chapter 2

Algebraisk spesifikasjon

I dette kapittelet skal vi studere spesifikasjon av abstrakte datatyper (ADT'er). En ADT er karakterisert som en mengde elementer, sammen med en mengde tilhørende funksjoner og konstanter. Vi betrakter i denne sammenheng en konstant som en funksjon som ikke tar argumenter. Det vil si at hvis T er en ADT, kan vi betrakte T som et par:

$$T == (V_T, F_T)$$

der V_T er mengden av elementer i T, og F_T er en mengde funksjoner som har typen T i sin signatur. Med signaturen til en funksjon mener vi her uttrykket:

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

der f er navnet til funksjonen, $T_1, \ldots, T_n$ er typene til alle argumentene til f, og T_{n+1} er typen til funksjonsverdien til funksjonen. Det er vanlig å dele F_T opp i to forskjellige klasser:

Generatorer : Dette er funksjoner som har typen T som funksjonsverdi. Hvis en generator ikke tar noen argumenter av type T sier vi at funksjonen er en *konstant.*

Observatører: Dette er en funksjon som tar ett eller flere argumenter av type T og returnerer et resultat som ikke er av type T.

Verdien til et element i en ADT blir bare indirekte spesifisert. Elementet kan f.eks. bli dannet ved gjentatte applikasjoner av typens generatorer på konstantene til typen. Samme element kan ofte genereres på mer enn en måte, dette blir spesifisert ved typen sine aksiomer, som relaterer funksjonene til hverandre. Siden en mengde av elementer, sammen med en samling funksjoner på elementene, danner en algebra, vil det være naturlig å betrakte en ADT som en algebra. For å spesifisere semantikken til en ADT kan vi derfor benytte algebraisk spesifikasjon.

2.1 Grunnleggende Begreper

I algebraisk spesifikasjon er det tre grunnleggende elementer :

- spesifikasjonen av en ADT.
- spesifikasjonens tilhørende aksiomer
- algebraene som tilfredstiller aksiomene

Spesifikasjonen av en ADT består av en mengde sorter og funksjoner, sammen med en mengde aksiomer. Aksiomene er logiske uttrykk som uttrykker oppførselen til funksjonene. Siden vi kun arbeider med ADT'er, bryr vi oss ikke om hvordan sortenes tilhørende elementer er representert. Det eneste som interesserer oss er egenskapene til funksjonene i ADT'en. Disse egenskapene blir beskrevet av aksiomene.

Som et eksempel på en algebraisk spesifikasjon av semantikken til en abstrakt datatype, kan vi spesifisere typen *Bool*. Denne typen har konstantene *sann* og *usann*, sammen med operasjonene '!' (ikke), '$\wedge$' (og), '$\vee$' (eller) og '$\Rightarrow$' (implikasjon):

```
specification BOOL
export
   sort         Bool
   operations
        sann   :              -> Bool
        usann  :              -> Bool
        ! _    : Bool         -> Bool
        _ /\ _ : Bool * Bool -> Bool
        _ \/ _ : Bool * Bool -> Bool
        _ => _ : Bool * Bool -> Bool
specifies
   variables    t, u : Bool
   equations
        ! sann      == usann
        ! usann     == sann
        t /\ sann   == t
        t /\ usann  == usann
        t /\ u      == u /\ t
        t \/ sann   == sann
        t \/ usann  == t
        t \/ u      == u \/ t
        t => u      == ( ! t ) \/ u
end specification
```

Spesifikasjonen har som overskrift nøkkelordet *specification*, og er etterfulgt av navnet til spesifikasjonen. Slutten av spesifikasjonen er markert med nøkkelordet *end specification*. Overskriften blir etterfulgt av signaturdelen, som består av sortene og funksjonene (med navn og type) til ADT'en vi skal spesifisere. Nøkkelordet *sorts* blir etterfulgt av sortene (typene) som blir spesifisert. Nøkkelordet *operations* lister alle funksjonene, sammen med deres signatur. Understrek blir brukt som en plassanviser sammen med navnet til funksjonene for å indikere hvor argumentene til funksjonene skal stå i forhold til funksjonsnavnet. Merk at vi kan spesifisere både infiks og prefiks notasjon på binære funksjoner. Vi ser derfor at '!' er en unær prefiks funksjon, og '$\wedge$' er binær og infiks. At funksjonene '*sann*' og '*usann*' ikke har noen plassanviser, betyr at de er konstanter. Det kan vi og se ut fra signaturdelen til funksjonene.

For hver sort er det en mengde velformede *termer*. Signaturen spesifiserer hvordan vi kan konstruere slike termer. Litt upresist kan vi si at:

- alle konstanter er termer.
- alle variabler er termer.
- anvendelsen av funksjoner på rett antall argumenter av rett sort er termer.

For eksempel er følgende gyldige termer i forhold til spesifikasjonen BOOL:

1. $sann$
2. u
3. $!\,sann$
4. $(t \wedge (!\,u)) \Rightarrow u$

Uttrykket '$\Rightarrow t$' er derimot ikke en term siden '$\Rightarrow$' er en binær funksjon. Termer som ikke inneholder noen variabler (f.eks. nr. 1 og nr. 3) kalles *grunntermer*. Når vi erstatter alle variabler i en term med grunntermer, blir resultatet og en grunnterm.

Resten av spesifikasjonen består av en beskrivelse av de abstakte egenskapene til ADT'en. Etter nøkkelordet *equations* kommer en samling av aksiomer.Aksiomene kan være betingede eller ubetingede likninger. En ubetinget likning er en likning mellom to termer av samme type. Variablene som inngår i likningene blir deklarert etter nøkkelordet *variables*. Alle variablene som inngår i likningene er implisitt universellt kvantorisert. Det betyr at alle likningene skal gjelde for alle mulige verdier for variablene som inngår. Hvis en likning er betinget, gjelder likningen for alle mulige verdier som oppfyller betingelsen for likningen. Likningene spesifiserer egenskapene til termene. F.eks. vil likningen $t \wedge sann = t$ kunne tolkes som 'for alle t gjelder det at '$t \wedge sann$' skal være lik t'. Dette kan beskrives mer formelt v.h.a. begrepene *grunnlikning* og *substitusjon*. En grunnlikning er en likning mellom grunntermer. Grunnlikninger kan konstrueres fra andre likninger ved å substituere alle variabler med grunntermer. Vi kan f.eks. danne følgende grunnlikninger:

- $sann \wedge sann == sann$
- $usann \wedge sann == usann$
- $!\, sann \wedge sann ==!\, sann$

Alle grunnlikningene over er dannet ved å substituere inn en grunnterm i likningen '$t \wedge sann = t$' i stedet for variabelen t. Hvis samme variabel inngår flere steder i samme likning, må alle forekomster substitueres med samme grunnterm, og grunntermen må være av samme sort som variabelen.

Til sammen vil alle likningene til en spesifikasjonen danne en logisk teori. Denne teorien definerer egenskapene som må eksistere hos enhver representasjon av vår spesifiserte ADT. Alle representasjoner som tilfredstiller vår teori kaller vi *modeller* av teorien. I vår spesifikasjon av typen BOOL, kan vi tenke oss flere representasjoner eller implementasjoner, men så lenge de tilfredstiller våre likninger, er de alle tilfredstillende modeller. Hvis vi benytter typen BOOL i et progam (eller i en annen spesifikasjon), er det vilkårlig hvilken modell som benyttes. Vi kan faktisk bytte representasjon, uten at programmet må endres. Det kan vi gjøre siden vi vet at alle modellene tilfredstiller vår teori for BOOL.

I eksempelet vårt med typen BOOL hadde vi kun en sort (Bool), sammen med en mengde funksjoner som kun hadde sorten Bool i sin signatur. Vi kan da uformellt si at alle modeller som tilfredstiller teorien for BOOL er en BOOL-algebra. Mer generellt kan en spesifikasjon definere mer enn en sort, og funksjonene i en spesifikasjon kan ha en signatur som består av mer enn en sort. Hvis funksjonene i en spesifikasjon benytter en sort som ikke blir definert i spesifikasjonen må den være importert. Det fører til at den importerte spesifikasjonen blir en del av spesifikasjonen som den blir inkludert inn i. Dette kan lett illustreres ved en spesifikasjon av positive heltall inkludert null. Siden heltall har en total ordning som avgjør om et tall er større enn et annet, trenger vi sorten Bool for å uttrykke relasjonen mellom to tall.

```
specification HELTALL
   include BOOL
export
   sort          Heltall
   operations
        0      :                     -> Heltall
        succ _ : Heltall             -> Heltall
        _ + _  : Heltall * Heltall -> Heltall
        _ < _  : Heltall * Heltall -> Bool
        _ er _ : Heltall * Heltall -> Bool
specifies
   variables    n, m : Heltall
   equations
```

```
        0 < 0                          ==  usann
        0 < ( succ 0 )                 ==  sann
        ( succ n ) < 0                 ==  usann
        ( succ n ) < ( succ m )        ==  n < m
        0 er 0                         ==  sann
        0 er ( succ n )                ==  usann
        ( succ n ) er ( succ m )       ==  n er m
        n er m                         ==  m er n
        0 + n                          ==  n
        ( succ n ) + m                 ==  succ ( n + m )
        n + m                          ==  m + n
end specification
```

Etter denne spesifikasjonen er alle grunntermer av sorten *Heltall* ekvivalent med enten konstanten 0 eller et endelig antall anvendelser av operasjonen *succ* på konstanten 0. F.eks. kan heltallet 3 skrives som *succ succ succ* 0. Det er og verd å merke seg at denne spesifikasjonen genererer nye grunntermer av typen Bool.

Det kan kanskje være nyttig å også innføre operasjonen *pred : Heltall* $\rightarrow$ *Heltall* som en invers funksjon av *succ*. Vi må da utvide HELTALL med likningene:

```
        pred 0                         == 0
        pred succ n                    == n
        pred ( n + m )                 == ( pred n ) + m
        ( pred  n ) < n                == sann
        ( pred  m ) < ( pred n )       == m < n
        ( pred  n ) er n               == usann
        ( pred  m ) er ( pred n )      == m er n
```

Men dette vil gi oss en teori som ikke er konsistent i forhold til hva vi vanligvis ønsker å uttrykke i vår formelle spesifikasjon. Det er fordi vi ved å bytte variabler med grunntermer kan utlede at *'sann = usann'* ! Dette kan lett vises med følgende eksempel:

$$\begin{array}{lcl} (pred\ n)\ er\ n & = & usann \\ (pred\ 0)\ er\ 0 & = & usann \\ 0\ er\ 0 & = & usann \\ sann & = & usann \end{array}$$

Denne motsigelsen strider mot vanlig fornuft, og gjør at sorten *Bool* kollapser til kun å inneholde ett element. Innenfor algebraisk spesifikasjon kalles dette fenomenet *forvirring* (eng. *confusion*). Det kan vises at en av grunnene til vår forvirring er likningen *pred* 0 $==$ 0, og den må derfor fjernes. Dette betyr igjen at vi ikke har noen likning som sier at *pred* 0 er lik et annet heltall, og må derfor også betraktes som et eget element i sorten *Heltall*. Dette er ikke med i vår forståelse av hva et positivt heltall er, og likningene over er derfor heller ikke nå tilfredstillende for å beskrive vårt system. Disse ekstra elementene (f.eks.

pred 0) er uønskede i forhold til vårt system, og kalles *søppel* (eng. *junk*). Dette korte eksempelet lærer oss flere ting:

1. Ved å gi en ny algebraisk spesifikasjon kan vi innføre forvirring i en allerede eksisterende spesifikasjon.
2. Man kan komme til å spesifisere elementer som ikke skal være med i sorten.

2.2 Parametriserte spesifikasjoner

Det er og mulig å gi en algebraisk spesifikasjon av *parametriserte typer*. En parametrisert type er en type som er satt sammen av en eller flere allerede eksisterende typer. Noen eksempler er par, lister og tabeller. Disse typene kallers også *generiske typer*. Spesifikasjonen består da av spesifikasjonsdelen beskrevet foran, sammen med en formell parameterdel. Denne delen inneholder en spesifikasjon over de formelle sortene, operasjonene og likningene som spesifikasjonen krever. En slik parametrisert spesifikasjon kan instantieres med aktuelle argumenter for så å danne en ordinær spesifikasjon. Et godt eksempel på en parametrisert spesifikasjon er spesifikasjonen for den generiske typen *Tabell*:

```
specification TABELL
   includes HELTALL
formal sort Element
exports
   sort        Tabell
   operations
      tabell  ( _, _ )     :Heltall * Element           -> Tabell
      les     ( _, _ )     :Heltall * Tabell            -> Element
      oppdater( _, _ , _ ) :Tabell  * Heltall * Element -> Tabell
      lengde  ( _)         :Tabell                      -> Heltall
specifies
   variables    i, j : Heltall
                e    : Element
                A    : Tabell
   equations

      lengde ( tabell ( i, e ) )      == i

      ( 0 < i <= lengde ( A ) ) =>
      lengde ( oppdater ( A, i, e ) ) == lengde ( A )

      ( 0 < j <= i ) =>
      les ( j , ( tabell ( i, e ) )   == e
```

```
    ( j  = i /\ 0 < i <= lengde ( A ) ) =>
    les ( j, oppdater ( A , i , e ) == e

    ( j <> i /\ 0 < i,j <= lengde ( A ) ) =>
    les ( j, oppdater ( A , i , e ) == les ( j, A )

end specification
```

Denne spesifikasjonen inneholder flere deler som ikke er brukt tidligere. Nøkkelordet *formal sort* sier at denne spesifikasjonen er parametrisert med sorten *Element*, og at vi ikke har noen endelig spesifikasjon før vi har instantiert spesifikasjonen TABELL med en ønsket sort. Flere av likningene er dessuten *betingede.* Det betyr at de kun gjelder hvis betingelsen er oppfyllt. Likningene er ikke helt fullstendige, men vi skal se mer på denne defininisjonen senere. Spesifikasjonen beskriver typen *Tabell* (av Element). Denne tabellen kan lettest betraktes som en indeksert sekvens av elementer, der indeksene er heltall i området 1 til '*lengde(A)*'.

For å instantiere denne spesifikasjonen, må vi ha en ny spesifikasjon som inkluderer spesifikasjonen TABELL. Et eksempel kan være å lage en tabell av heltall :

```
specification HELTALL_TABELL
   include HELTALL

   include instantiation TABELL by HELTALL
      using          Heltall       for Element
      renamed using  heltallTabell for Tabell
end specification
```

Nå er spesifikasjonen HELTALL_TABELL en instantiering av spesifikasjonen TABELL, der TABELL bruker sorten *Heltall* for *Element.* Nøkkelordet *using* benytter vi for å binde TABELL sine formelle argumenter (det kan være sorter, funksjoner eller likninger) til de aktuelle parameter vi ønsker å bruke. Det kan være at vi kanskje senere også ønsker å instantiere spesifikasjonen TABELL med andre aktuelle argumenter, for eksempel sorten *Bool.* For å unngå overlasting av sortsnavn er det derfor lurt å gi den instantierte sorten et nytt og unikt navn. Det gjør vi ved først å spesifisere nøkkelordet *renamed using*, og etterpå gi en liste av nye navn som skal erstatte de eksisterende navnene fra instantieringen.

2.3 Spesifikasjoner som aksiom

Det er hender ofte at flere typer har en eller flere like operasjoner, d.v.s. operasjoner med samme signatur og algebraiske likninger. Et naturlig tilfelle er likheten mellom heltall og reelle tall m.h.p. operasjonen '+'. Vi vet at begge typene inneholder aksiomet $a + (b + c) = (a + b) + c$. Dette aksiomet benevnes *asossiativitet.* Vi ønsker ofte å uttrykke at en sort og en operasjon har en spesiell egenskap som mange andre sorter og typer har, og vi innfører derfor følgende notasjon:

```
specification ASOSSIATIVITET
parameters
   sort        t
   operation   _ * _ : t * t -> t
specifies
   variables   a, b, c : t
   equation    a * ( b * c ) = ( a * b ) * c
end specification
```

Dette er kun en videreføring av det som ble sagt om algebraisk spesifikasjon tidligere, og det skjer ved at vi innfører nøkkelordet *parameters*. Dette nøkkelordet sier at hele spesifikasjonen skal betraktes som et aksiom over de aktuelle argumentene som spesifikasjonen blir instantiert med. Alle likningene i spesifikasjonen uttaler seg om de bundne parametrene ved en eventuell instantiering. Siden vi bruker denne spesifikasjonen som et aksiom (i spesifikasjonsdelen til den inkluderende spesifikasjon) kan den ikke ta formelle parametre, og den kan heller ikke eksportere noen sorter eller operasjoner.

2.4 Noen utvidede spesifikasjoner

Spesifikasjonene over er formellt korrekte, men spesifikasjonene for HELTALL og TABELL er litt begrensede. Jeg vil derfor her gi en utvidet algebraisk spesifikasjon, og jeg kommer til å benytte disse som komponenter i alle senere spesifikasjoner som inkluderer HELTALL og TABELL. Jeg vil først og fremst utvide heltallene til og å bestå av negative heltall, og jeg ønsker å innføre den mer naturlige notasjonen '[ ]' for å indeksere elementer i en tabell.

```
specification HELTALL
include BOOL
export
   sort        Heltall
   operations
      0      :                       ->   Heltall
      succ _ : Heltall               ->   Heltall
      pred _ : Heltall               ->   Heltall

      - _    : Heltall               ->   Heltall
      _ + _  : Heltall * Heltall     ->   Heltall
      _ * _  : Heltall * Heltall     ->   Heltall
      _ - _  : Heltall * Heltall     ->   Heltall
```

```
      _ < _   : Heltall * Heltall   ->   Bool
      _ = _   : Heltall * Heltall   ->   Bool
      _ > _   : Heltall * Heltall   ->   Bool
      _ <= _  : Heltall * Heltall   ->   Bool
specifies
   variables    m, n, p : Heltall
   equations
      pred succ n = n
      succ pred n = n
      ASOSSIATIVITET ( Heltall, + )
      ASOSSIATIVITET ( Heltall, * )
      n + m              ==   m + n
      n + succ m         ==   succ ( n + m )
      n + 0              ==   n
      n + pred m         ==   pred ( n + m )
      n + ( - n )        ==   0
      n - m              ==   n + ( - m   )
      n * m              ==   m * n
      n * succ m         ==   n + n * m
      n * 0              ==   0
      n * ( m + p )      ==   n * m + n * p
      n     < n          ==   usann
      pred n < n         ==   sann
      n     < succ n     ==   sann
      pred n < pred m    ==   n < m
      succ n < succ m    ==   n < m
      n > m              ==   m < n
      ( n = m )          ==   ! ( n < m ) /\ ! ( m < n )
      n <= m             ==   ! ( n > m )
end specification
```

I denne spesifikasjonen er det flere ting å merke seg. Det første er at vi benytter spesifikasjonen for ASOSSIATIVITET. Videre er det nyttig å merke seg at operasjonene '−' og '∗' er entydig definert av likningene for '+'. Vi ser og at operasjonen '<' bestemmer oppførselen til '=' og '>'.

```
specification TABELL
   includes BOOL
   includes HELTALL
formal
   sort Element
exports
   sort Tabell
```

```
  operations
     tabell ( _, _ )     : Heltall * Element
                           | ( 0 < #1 )              -> Tabell
     lengde ( _ )        : Tabell                    -> Heltall
     okIndeks ( _, _ )   : Tabell * Heltall          -> Bool
     _ [ _ ]             : Tabell * Heltall
                           | okIndeks ( #1, #2 )     -> Element
     _ [ _ ] := _        : Tabell * Heltall * Element
                           | okIndeks ( #1, #2 )     -> Tabell
specifies
  variables     i, j : Heltall
                e    : Element
                A    : Tabell
  equations
     okIndeks ( A, i )          == ( 0 < i ) /\ ( i <= lengde ( A )
     lengde ( tabell ( i, e ) ) == i
     lengde ( A [ i ] := e )    == lengde ( A )
     tabell ( i, e )   [ j ]    == e

         ( j = i )   =>  ( A [ i ] := e )  [ j ]     == e
     ( ! ( j = i ) ) =>  ( A [ i ] := e )  [ j ]     == A [ j ]
end specification
```

I spesifikasjonen for TABELL har jeg utvidet notasjonen noe. Jeg benytter predikatfunksjonen *okIndeks* i signaturdelen til operasjonene for å begrense lovlige argumenter til operasjonene. Det gjør at kun argumenter som tilfredstiller predikatet er gyldige argumenter til den voktede funksjonen. Argumentene til en predikatfunksjon er en delmengde av de argumentene som blir gitt til operasjonen som predikatet skal 'beskytte'. Siden det kun er en delmengde av argumentene som benyttes i predikatet, benytter jeg meg av notasjonen '$\#i$' for å angi at argument nr. i til den beskyttede operasjonen skal inngå i predikatet.

Hvis vi skal uttrykke at lengden til en tabell skal være større enn null må vi, hvis vi skal være helt formelle, skrive følgende aksiom:

```
( 0 < lengde ( A ) ) == sann
```

Vi skal videre være litt uformelle i med hensyn på sorten *Bool* i våre algebraiske spesifikasjoner, og bare skrive:

```
0 < lengde ( A )
```

Dette betyr at alle aksiomer av sorten *Bool* implisitt skal være lik den boolske konstanten *sann*.

Chapter 3

Ringer, moduler og komoduler

Det er svært mange likheter mellom algebraisk spesifikasjon og vanlig matematisk formalisme. Som vi etter hvert vil se, er det mye implisitt informasjon i matematiske definisjoner. Dette er informasjon som må gjøres eksplisitt i enhver formell spesifikasjon. Det som særlig er implisitt er overlastingen av operatorer (bruk av flere funksjoner med samme navn, men med forskjellig signatur). F.eks. er funksjonen '+' mellom heltall ikke den samme som '+' mellom reelle tall. Noe annet som nesten alltid er implisitt, er konvertering mellom to typer. Den vanligste konverteringen er antageligvis fra heltall til reelle tall.

Det er argumenter både for og mot bruk av overlasting av funksjoner, men jeg velger i min fremstilling å benytte overlasting av funksjoner og konstanter når de representerer matematiske begreper som vanligvis er overlastet i matematisk litteratur. Jeg ønsker derimot ikke å benytte implisitt typekonvertering, da dette sjelden er nødvendig. Jeg vil heller konvertere typene eksplisitt de få stedene de forekommer.

3.1 Ringer

3.1.1 Matematisk definisjon

En ring er et trippel $\langle K, +, * \rangle$, der K er en mengde av elementer, og '$+ : K * K \to K$' og '$* : K * K \to K$' er funksjoner som oppfyller følgende krav:

$$\begin{array}{llll}
\mathbf{K1}: & \exists\, 0 \in K,\ \forall a \in K & : a + 0 & = a \\
\mathbf{K2}: & \forall a, b \in K & : a + b & = b + a \\
\mathbf{K3}: & \forall a, b, c \in K & : a + (b + c) & = (a + b) + c \\
\mathbf{K4}: & \forall a \in K,\ \exists - a \in K & : a + -a & = 0 \\
\mathbf{K5}: & \exists 1 \in K,\ \forall a \in K & : a * 1 & = a \\
\mathbf{K6}: & \forall a, b, c \in K & : a * (b * c) & = (a * b) * c \\
\mathbf{K7}: & \forall a, b, c \in K & : (b + c) * a & = (b * a) + (c * a) \\
\mathbf{K8}: & \forall a, b, c \in K & : a * (b + c) & = (a * b) + (a * c)
\end{array}$$

□

Dette er en svært generell definisjon av en ring, men det er ganske vanlig å betrakte mengden K som en delmengde av de komplekse tall.

3.1.2 Algebraisk spesifikasjon

Definisjonen over kan lett skrives om til en algebraisk spesifikasjon :

```
specification RING
parameters
   sort       Ring
   operations
      0     :              -> Ring
      1     :              -> Ring
      - _   : Ring         -> Ring
      _ + _ : Ring * Ring  -> Ring
      _ * _ : Ring * Ring  -> Ring
specifies
   variables a, b, c : Ring
   equations
      a + 0             ==  a                        //  K1
      a + b             ==  b + a                    //  K2
      a + ( b + c )     ==  ( a + b ) + c            //  K3
      a + ( - a )       ==  0                        //  K4
      a * 1             ==  a                        //  K5
      a * ( b * c )     ==  ( a * b ) * c            //  K6
      ( b + c ) * a     ==  ( b * a ) + ( c * a )    //  K7
      a * ( b + c )     ==  ( a * b ) + ( a * c )    //  K8
end specification
```

Det er verdt å merke seg at alle elementer og funksjoner som er eksistensiellt kvantorisert i den matematiske definisjonen er tatt med som funksjoner i den algebraiske spesifikasjonen. Denne metoden kalles ofte *Skolemisering*. Alle elementer som er universellt kvantorisert, er derimot satt opp etter nøkkelordet *variables*, og inngår direkte i likningene for spesifikasjonen. Merk og at det er variablene i hver enkelt aksiom som er allkvantorisert, ikke kombinasjonene av alle aksiomene. Dette betyr at alle variablene som inngår i et aksiom er fri for restriksjoner fra de andre aksiomene.

Merk at vi her bruker den algebraiske spesifikasjonen som et aksiom. Det er siden spesifikasjonen ikke definerer noen sorter eller funksjoner. Vi kan da bruke denne spesifikasjonen til å påstå ringegenskaper for allerede eksisterende sorter og funksjoner. Et eksempel på bruk av vår algebraiske spesifikasjon av en ring kan da være å definere en *Kropp*:

```
specification KROPP
```

```
parameters
   sort       Kropp
   operations
      0     :                    ->  Kropp
      1     :                    ->  Kropp
      - _   : Kropp              ->  Kropp
      _ + _ : Kropp * Kropp      ->  Kropp
      _ * _ : Kropp * Kropp      ->  Kropp
      1 / _ : Kropp ( #1 != 0 )  ->  Kropp
specifies
   variables a, b : Kropp
   equations
      RING ( Kropp, 0, 1, -, +, * )
      a * b       == b * a
      a * ( 1/a ) == 1
end specification
```

En av fordelene med å omformulere de matematiske definisjonene til en algebraisk spesifikasjon er at den algebraiske spesifikasjonen gir oss en struktur som lett kan oversettes til et programmeringsspråk. Hvis vi ser bort fra noen mindre programmeringsdetaljer, kan vi lett 'oversette' spesifikasjonen til en C++ klasse signatur:

```
class Ring
{
   public :
static Ring    null        (              )       ;
static Ring    en          (              )       ;
       Ring    operator -  (              ) const ;
       Ring    operator +  ( const Ring & ) const ;
       Ring    operator *  ( const Ring & ) const ;
} ;
```

Husk her her på at første argument til en funksjon i en C++ klasse implisitt er et objekt i klassen. Typen til dette argumentet er derfor ikke tatt med i signaturen til funksjonene. Merk og nøkkelordet *static*. Det forteller at en funksjon skal 'deles' mellom alle objektene i en klasse. Man trenger derfor ikke et klasseobjekt for å kalle funksjonene '*null*' og '*en*. Disse funksjonene kan i stedet kalles slik som '`Ring::null()`'. I C++ har vi dessuten den fordelen at vi kan overlaste alle operatorene i språket. Det betyr at hvis vi har en modell som tilfredstiller spesifikasjonen RING, kan modellen implementeres som en klasse i C++ med en signatur som over. Vi kan senere skrive programmer som har et naturlig matematisk uttrykk. Et lite eksempel er:

```
Ring r1 = Ring::null() ;
```

```
Ring r2 = Ring::en  () ;
Ring r3                ;

r3 = r2 + r2 ;
```

3.2 Moduler

En mengde V og en ring $\langle K, +, *\rangle$, sammen med funksjonene '$+ : V * V \rightarrow V$' og '$* : K * V \rightarrow V$' kalles en *modul* over ringen K, hvis følgende betingelser er oppfyllt:

$$
\begin{array}{llcll}
\mathbf{M1}: & \exists 0 \in M, ,\forall a \in M & : & a+0 & = & a \\
\mathbf{M2}: & \forall a \in M ,\exists -a \in M & : & a+(-a) & = & 0 \\
\mathbf{M3}: & \forall a,b,c \in M & : & a+(b+c) & = & (a+b)+c \\
\mathbf{M4}: & \forall a,b \in M & : & a+b & = & b+a \\
\mathbf{M5}: & \forall c \in K ,\forall a,b \in M & : & c*(a+b) & = & c*a+c*b \\
\mathbf{M6}: & \forall c,k \in K ,\forall a \in M & : & (c+k)*a & = & c*a+k*a
\end{array}
$$

Vi benevner modulen $\langle V, K, +, *\rangle$. □

3.2.1 Algebraisk spesifikasjon

Vi ser at definisjonen for en modul har den samme matematiske struktur som for en ring, og vi kan derfor gi en algebraisk spesifikasjon mye på samme måte som tidligere :

```
specification MODUL
parameters
   sort          Ring, Modul
   operations
      0          :                 -> Ring
      1          :                 -> Ring
      - _        : Ring            -> Ring
      _ + _      : Ring * Ring     -> Ring
      _ * _      : Ring * Ring     -> Ring

      0          :                 -> Modul
      _ + _      : Modul * Modul   -> Modul
      - _        : Modul           -> Modul
      _ * _      : Ring * Modul    -> Modul
specifies
   variables    m, n    : Ring
                a, b, c : Modul
   equations
      RING ( Ring, 0, 1, -, +, * )
```

```
    a + 0           ==  a                //  M1
    a + ( - a )     ==  0                //  M2
    a + ( b + c )   ==  ( a + b ) + c    //  M3
    a + b           ==  b + a            //  M4
    m * ( a + b )   ==  m * a + m * b    //  M5
    ( m + n ) * a   ==  m * a + n * a    //  M6
end specification
```

Merk at sorten *Modul* ikke oppfyller spesifikasjonen MODUL hvis ikke sorten *Ring* oppfyller spesifikasjonen RING. Vi må derfor parametrisere denne spesifikasjonen med sorten *Ring* og alle sortens operasjoner slik at vi kan verifisere ringegenskapene.

Det er også i dette tilfellet lett å oversette den algebraiske spesifikasjonen til et programmeringsspråk. En kan for eksempel gi følgende klassesignatur:

```
class modul
{  public:

static modul  null        (                  )       ;
       modul  operator  - (                  ) const ;
       modul  operator  + ( const modul &    ) const ;
friend modul  operator  * ( const ring  & ,
                            const modul &    )       ;
} ;
```

der klassen *ring* må ha samme klassesignatur som tidligere spesifisert. Hvis klassen *ring* oppfyller spesifikasjonen RING og klassen *modul* oppfyller spesifikasjonen MODUL, er klassen en modell av spesifikasjonen. Merk her nøkkelordet *friend*. Siden operasjonen '*' skal ha et venstreargument fra en ringklasse, kan ikke operasjonen '*' være en funksjon i klassen *modul*. Vi må derfor deklarere '*' som en *friend* til klassen *modul*. Operasjonen '*' vil da få tilgang til datastrukturen til modulklassen, og vil derfor kunne oppføre seg som de andre medlemsfunksjonene.

3.3 komoduler

3.3.1 Lineære avbildninger

La E og F være to moduler over samme ring K. En avbildning '$A : E \rightarrow F$' er lineær dersom følgende krav er oppfyllt:

L1 : $\forall x, y \in E : A(x+y) = A(x) + A(y)$

L2 : $\forall c \in K,\ \forall x \in E : A(c * x) = c * A(x)$

Mengden av alle lineære avbildninger fra en modul E til en modul F betegnes $L(E;F)$.□

En lineær avbildning $L(E;F)$ får modulegenskaper hvis vi utruster avbildningen med operasjonene '0', '+', '-' og '*' med følgende egenskaper:

L3 : $\forall m \in E : 0(m) == 0$

L4 : $\forall A, B \in$L(E;F), $\forall m \in E : (A+B)(m) = A(m) + B(m)$

L5 : $\forall A \in$L(E;F), $\forall m \in E : (-A)(m) = -A(m)$

L6 : $\forall A \in$L(E;F) ,$\forall m \in E$, $\forall k \in K : (k * A)(m) = k * A(m)$

3.3.2 Duale moduler

Gitt en modul V over en ring K. Da kaller vi mengden av alle lineære avbildninger fra V til K den *duale modul* til V, og benevner den V^*. D.v.s. :

$$V^* = L(V;K)$$

Vi sier ofte at V^* er en *komodul* til V.□

3.3.3 Algebraisk spesifikasjon

Vi vil nå gi en algebraisk spesifikasjon av komoduler. Siden en dual modul V^* er sterkt knyttet opp mot en modul V kan vi kun spesifisere V^* i forhold til V:

```
specification KOMODUL
parameters
   sort          Ring, Modul, koModul
   operations
      0        :                      -> Ring
      1        :                      -> Ring
      - _      : Ring                 -> Ring
      _ + _    : Ring * Ring          -> Ring
      _ * _    : Ring * Ring          -> Ring

      0        :                      -> Modul
      _ + _    : Modul * Modul        -> Modul
      - _      : Modul                -> Modul
      _ * _    : Ring * Modul         -> Modul

      0        :                      -> koModul
      _ + _    : koModul * koModul    -> koModul
      - _      : koModul              -> koModul
```

```
      _ * _     : Ring * koModul      -> koModul
      _ ( _ )   : koModul * Modul     -> ring
specifies
   variables m, n : Modul
             a, b : koModul
             k    : Ring
   equations
      RING     ( Ring, 0, 1, -, +, *    )
      MODUL    ( Ring, Modul, 0, 1, -, +, * ,0, +, -, * )
      MODUL    ( Ring, koModul, 0, 1, -, +, * ,0, +, -, * )

      a ( m + n )        ==   a ( m ) + a ( n )     \\  L1
      a ( k * m )        ==   k * a ( m )           \\  L2
      0 ( m     )        ==   0                     \\  L3
      ( a + b ) ( m )    ==   a ( m ) + b ( m )     \\  L4
      ( - a   ) ( m )    ==   - ( a ( m ) )         \\  L5
      ( k * a ) ( m )    ==   k * a ( m )           \\  L6
end specification
```

Her krever vi at sorten *Ring* tilfredstiller spesifikasjonen RING, sorten *Modul* tilfredstiller sorten MODUL, og at en dual modul og skal tilfredstille spesifikasjonen MODUL. I tillegg skal kravene om linearitet gjelde.

Vi har også her en klar sammenheng mellom den algebraiske spesifikasjonen og mulige implementasjoner i et programmeringsspråk. Se for eksempel på følgende C++ signatur:

```
class koModul
{ public:

static koModul  null         (                    )       ;
       koModul  operator  -  (                    ) const ;
       koModul  operator  +  ( const koModul &    ) const ;
friend koModul  operator  *  ( const Ring    &  ,
                               const koModul &    )       ;
       Ring     operator  () ( const Modul &      ) const ;
} ;
```

Merk at denne klassen har en signatur som inneholder alle funksjonene i klassen *Modul*.

3.4 Viktige egenskaper ved moduler og komoduler

Moduler og komoduler har en del nyttige egenskaper som vi kan utlede fra definisjonene over. Disse egenskapene trenger vi når vi senere skal implementere noen modeller av spesifikasjonene. Vi må derfor se nærmere på hvilke egenskaper moduler har, og hvilke muligheter disse egenskapene gir oss med hensyn på eventuelle implementasjoner. Det som er mest aktuellt for oss, er å se på begrepene *basis* og *koordinater*.

3.4.1 Lineær kombinasjon

Gitt en modul $\langle V, K, +, *\rangle$. En endelig sum av typen:

$$c^1 * \mathbf{v_1} + ... + c^k * \mathbf{v_k}$$

der $\mathbf{v_i} \in V$ og $c^i \in K$, kalles en *lineær kombinasjon* av elementene $\{\mathbf{v_1}, \ldots, \mathbf{v_k}\}$. $\square$

3.4.2 Lineær uavhengighet

Gitt en modul $\langle V, K, +, *\rangle$ og en mengde $M \subseteq V$. Vi sier at alle elementene i M er *lineært uavhengige* dersom for alle utvalg $\{\mathbf{v_1}, \ldots, \mathbf{v_k}\} \subseteq M (k < \infty)$, og for alle $c^1, \ldots, c^k \in K$ er likningen :

$$c^1 * \mathbf{v_1} + \ldots + c^k * \mathbf{v_k} = \mathbf{0}$$

oppfyllt hvis og bare hvis $c^i = 0$ $(0 < i \leq k)$. $\square$

3.4.3 Basis

Gitt en modul $\langle V, K, +, *\rangle$. En mengde $B \subseteq V$ kalles en *basis* for V dersom følgende betingelser er oppfylt:

B1 : Alle elementer $\mathbf{b_i} \in B$ er lineært uavhengige.

B2 : Alle elementer $\mathbf{v} \in V$ kan skrives som en lineær kombinasjon av elementene i B.

$\square$

Antallet elementer i B kaller vi *dimensjonen* til modulen. Hvis B er en endelig mengde, sier vi at modulen er endelig dimensjonal. Ellers sier vi at modulen er uendelig dimensjonal.

3.4.4 Koordinater

Hvis en endelig dimensjonal modul $\langle V, K, +, *\rangle$ har en basis $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$, kan alle elementer $\mathbf{v} \in V$ skrives entydig som en lineær kombinasjon:

$$v = c^1 * \mathbf{b_1} + \ldots + c^d * \mathbf{b_d}, \quad der \;\; c^i \in K, 0 < i \leq d$$

Vi sier da at sekvensen $\langle c^1, \ldots, c^d\rangle$ er *koordinatene* til $\mathbf{v}$ m.h.p. basisen B. $\square$

I det etterfølgende vil vi kun arbeide med moduler som er endelig dimensjonale. Det betyr at alle moduler vil ha en mengde av elementer som danner en basis for modulen, og at alle elementer derfor kan uttrykkes som en lineær kombinasjon av denne basisen. I stedet for å benevne en modul som $\langle V, K, +, *\rangle$, skal vi være litt upresise og bare si at vi har en modul V over en ring K. Hvis vi har gitt en modul V over en ring K, kan vi alltid konstruere en ny dual modul V^*ved hjelp av følgende definisjon:

Definisjon

La V være en modul over en ring K, og la V ha dimensjon d. Da har den duale modul V^* og dimensjon d. Hvis $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ er en basis for V, og mengden $F = \{\mathbf{f^1}, \ldots, \mathbf{f^d}\} \subseteq V^*$ har følgende egenskap:

$$\mathbf{f^i}(\mathbf{b_j}) = \delta^i_j = \begin{cases} 0 & \text{hvis } i <> j \\ 1 & \text{hvis } i = j \end{cases}$$

er F en basis for V^*, og kalles den *duale basis* til B.□

3.4.5 Algoritmer og datastruktur

Egenskapene til moduler er svært viktige for oss siden vi nå vet hvordan vi kan implementere alle vanlige moduloperasjoner. La V være en modul over en ring R med basis $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$. La videre $v \in V$ være et element som kan skrives som den lineære kombinasjonen $\mathbf{v} = v^i * \mathbf{b_i}$, og la $f \in V$ skrives som $\mathbf{f} = f^i * \mathbf{b_i}$. Da kan det lett bevises at følgende ekvivalenser gjelder:

- $\mathbf{0} == \sum_{i=1}^d 0 * \mathbf{b_i}$
- $\mathbf{e} + \mathbf{f} == \sum_{i=1}^d (e^i + f^i) * \mathbf{b_i}$
- $(-\mathbf{e}) == \sum_{i=1}^d (-e^i) * \mathbf{b_i}$
- $k * \mathbf{f} == \sum_{i=1}^d (k * f^i) * \mathbf{b_i}$

for alle $k \in R$, og $\mathbf{0} \in V$ er et nullelement til modulen V. Dette betyr at vi ikke trenger implementere f.eks. addisjon mellom to elementer i mengden V, siden vi bare trenger addere alle koordinatene til elementene.

Vi kan og utlede hvordan vi beregner resultatet av en anvendelse av et modulelement på et element i en komodul med dual basis. La V være en modul over ringen R med basis $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$. La videre V^* ha den duale basis $B' = \{\mathbf{b^1}, \ldots, \mathbf{b^d}\}$. Siden et element $\mathbf{v} \in V$ kan skrives som en lineær kombinasjon $\mathbf{v} = v^1 * \mathbf{b_1} + \ldots + v^d * \mathbf{b_d}$, og et element $\mathbf{k} \in V^*$ kan skrives som $\mathbf{k} = k_1 * \mathbf{b^1} + \ldots + k_d * \mathbf{b^d}$, kan vi beregne anvendelsen $\mathbf{k}(\mathbf{v})$ på følgende måte:

$$\mathbf{k}(\mathbf{v}) = \sum_{i=1}^{d}\sum_{j=1}^{d}(k_i * \mathbf{b}^\mathbf{i})(v^j * \mathbf{b_j}) = \sum_{i=1}^{d}\sum_{j=1}^{d} k_i * v^j * \mathbf{b}^\mathbf{i}(\mathbf{b_j}) = \sum_{i=1}^{d}\sum_{j=1}^{d} k_i * v^j * \delta^i_j = \sum_{i=1}^{d} k_i * v^i$$

Dette betyr at anvendelsen kan beregnes selv om vi kun har tilgang til koordinatene til elementene, og at vi nå har alle de algoritmene vi trenger for å implementere alle mulige moduler og komoduler over en ring.

Fra nå av skal vi bruke regelen at det alltid skal utføres en implisitt summasjon over alle repeterte indekser i et uttrykk. Det fører til at beregningen over skrives på følgende form:

$$\mathbf{k}(\mathbf{v}) = (k_i * \mathbf{b}^\mathbf{i})(v^j * \mathbf{b_j}) = k_i * v^j * \mathbf{b}^\mathbf{i}(\mathbf{b_j}) = k_i * v^j * \delta^i_j = k_i * v^i$$

Siden vi gir en modul med d dimensjoner en fast basis, trenger vi kun lagre koordinatene til elementene i modulen. Disse elementene kan lett lagres i tabeller som inneholder d ring-elementer.

3.5 Utvidelse av spesifikasjonen

Vi skal videre kun arbeide med moduler som har en fast basis. Dette betyr alle elementer i en modul kan skrives som en lineær kombinasjon av en sekvens koordinater og den faste basisen. For å få aksess til koordinatene og basisen til et modulelement må vi derfor utvide spesifikasjonen til og å inneholde følgende operasjoner:

```
dimensjon ( _ )   : Modul                                    -> Heltall
koordinat ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
basis     ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1)) -> Modul
```

der operasjonen *dimensjon (e)* returnerer antallet basiselementer som modulen inneholder. Operasjonen *basis (e, i)* returnerer basiselement nummer i til modulen, og operasjonen *koordinat (e, i)* returnerer koordinat nummer i til elementet e. Vi må nå gi spesifikasjonen noen aksiomer som uttrykker den uformelle beskrivelsen. Det vil si at vi må vise at:

- En modul har minst ett basiselement.
- Alle elementer i modulen har samme basis.
- Alle basiselementene er lineært uavhengige.
- Den lineære kombinasjonen av koordinatene til et element e og basisen gir oss som resultat elementet e.

Dette kan gjøres på følgende måte (vi tar bare med et utsnitt av spesifikasjonen):

```
uses
   linkomb ( _,_ ) : Heltall * Modul | (0<=#1<=dimensjon(#2))  -> Modul
specifies
   variables  i, j : Heltall
              e, f : Modul
   equations
      ( 0 < dimensjon ( e )          == sann
      dimensjon ( e )                == dimensjon ( f )
      basis ( e, i )                 == basis ( f , i )
      linKomb ( dimensjon ( e ), e ) == e

      ( i  = j ) => koordinat ( basis ( 0, i ) , j ) == 1
      ( i <> j ) => koordinat ( basis ( 0, i ) , j ) == 0

      linKomb ( 1, e ) == koordinat ( e, 1 ) * basis ( e, 1 )
      linKomb ( i, e ) == koordinat ( e, i ) * basis ( e, i ) +
                          linKomb ( i - 1, e )
end specification
```

Her bruker vi en hjelpefunksjon *linKomb*, som summerer opp produktet mellom koordinatene til et element og basisen. Vi krever i spesifikasjonen at antall dimensjoner skal være mer enn null, den lineære kombinasjonen av koordinatene til et element og basisen skal være elementet selv, og at et basiselement ikke skal kunne uttrykkes som en lineær kombinasjon av noen andre basiselementer. Merk og at flere av operasjonene er partielle (d.v.s. at de er definert kun for de argumentene som oppfyller predikatet som vokter funksjonene). Siden en dual modul og er en modul, må spesifikasjonen KOMODUL også utvides med operasjonene over.

Vi skal videre utvide spesifikasjonen KOMODUL til å inneholde en spesifikasjon av *dual basis*. Vi må da kreve at operasjonen '_(_)' tilfredstiller definisjonen av dual basis over. Det vil si at vi trenger likningene:

```
variables
   m    : koModul
   e    : Modul
   i, j : Heltall
equations
   ( i  = j ) =>  basis ( m, i ) ( basis ( e, j ) ) == 1
   ( i <> j ) =>  basis ( m, i ) ( basis ( e, j ) ) == 0
```

3.6 Skifte av basis

Det finnes flere måter å definere moduler og komoduler over en ring. I denne oppgaven benytter vi en forholdsvis abstrakt definisjon. Det finnes andre definisjoner av moduler som baserer seg på hvordan koordinatene til et modulelement vil endre seg hvis en endrer basisen (se f.eks. [4]). Disse definisjonene er ekvivalente, og det kan være nyttig å vise dette. Vi skal ikke benytte noen av resultatene i dette avsnittet senere i oppgaven, men avsnittet kan likevel være nyttig siden vi her finner en del interessante egenskaper ved moduler og lineære transformasjoner.

Gitt en modul $\langle V, K, +, *\rangle$ med en basis $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$. Alle elementer $\mathbf{v} \in V$ kan nå skrives som en lineær kombinasjon av koordinatene $\{v^1, \ldots, v^d\}$ og basisen B på følgende måte:

$$v = v^1 * \mathbf{b_1} + \ldots + v^d * \mathbf{b_d}$$

Et element $\mathbf{k} \in V^*$ kan også representeres som en lineær kombinasjon av den duale basisen $K = \{\mathbf{k^1}, \ldots, \mathbf{k^d}\}$.

Det kan generellt eksistere mange forskjellige basiser for V. La oss derfor bytte basis for V til basisen $B' = \{\mathbf{b'_1}, \ldots, \mathbf{b'_d}\}$. Siden alle nye basiselementer er elementer i V kan vi uttrykke den nye basisen som en lineær kombinasjon:

$$\mathbf{b'_j} = a^i_j * \mathbf{b_i}$$

der a^i_j er koordinatene til $\mathbf{b'_j}$ med hensyn til basisen B.

3.6.1 Endringer for den duale basis

Hvis en modul V får en ny basis B', må V^* få en ny dual basis K'. Dette er siden vi pr. definisjon må kreve at:

$$\mathbf{k^i}(\mathbf{b_j}) = \mathbf{k'^i}(\mathbf{b'_j}) = \delta^i_j$$

Den nye duale basisen K' må vi og kunne finne ved en lineær kombinasjon::

$$\mathbf{k'^j} = p^j_i * \mathbf{k^i}$$

der p^j_i er koordinatene til $\mathbf{k'^j}$ med hensyn til den gamle duale basisen K. Vi kan nå finne koordinatene p^j_i på følgende måte:

$$\begin{aligned}
\mathbf{k'^i}(\mathbf{b'_j}) &= \delta^i_j \\
p^i_m * \mathbf{k^m}(a^n_j * \mathbf{b_n}) &= \delta^i_j \\
p^i_m * a^n_j * \mathbf{k^m}(\mathbf{b_n}) &= \delta^i_j \\
p^i_m * a^n_j * \delta^m_n &= \delta^i_j \\
p^i_m * a^m_j &= \delta^i_j \\
p^i_m &= (a^{-1})^i_m
\end{aligned}$$

der vi med uttrykket $(a^{-1})^i_j$ mener mengden av koordinater som tilfredstiller likningen $(a^{-1})^i_j * a^j_p = \delta^i_p$. Dette betyr at den duale basisen F transformerer som følgende:

$$\mathbf{k}'^{\mathbf{i}} = (a^{-1})^i_j * \mathbf{k}^{\mathbf{j}}$$

3.6.2 Endringer for koordinatene til et modulelement

Et skifte av basis for V vil føre til at vi også må utføre et skifte av koordinatene til alle elementene i V. Det er siden et element $\mathbf{v} \in V$ er representert som en lineær kombinasjon av koordinatene $\{c^1, \ldots, c^n\}$ og basisen B uansett hvilken basis vi bruker. Hvis vi skifter til ny basis B' må vi og finne de nye koordinatene som tilfredstiller:

$$\mathbf{v} = c^i * \mathbf{b_i} = c'^i * \mathbf{b'_i}$$

Vi kan utlede transformasjonsfaktorene vi trenger for å finne de nye koordinatene C':

$$\begin{aligned}
\mathbf{v} &= c^i * \mathbf{b_i} \\
\mathbf{v} &= c^i * (A^{-1})^j_i * \mathbf{b'_j} \\
\mathbf{v} &= c'^j * \mathbf{b'_j} \\
c'^j &= c^i * (A^{-1})^j_i *
\end{aligned}$$

Merk at koordinatene til et element $\mathbf{v}$ transformerer 'motsatt' av hvordan basiselementene transformeres. Det er derfor vanlig å si at koordinatene transformerer *kontravariant* i forhold til B. Dette markerer vi ved å indeksere alle koordinatene til et modulelement med hevet indeks. Siden elementene i B transformerer *kovariant* i forhold til seg selv indekserer vi basiselementene med senket indeks.

3.6.3 Endringer for koordinatene til et komodulelement

Når vi har funnet hvordan den duale basis transformeres m.h.p. bytte av basis for V, kan vi lett finne hvordan koordinatene til et element $\mathbf{e} \in V^*$ vil transformeres:

$$\begin{aligned}
\mathbf{e} &= f_i * \mathbf{k^i} \\
\mathbf{e} &= f_i * A^i_j * \mathbf{k'^j} \\
\mathbf{e} &= f'_j * \mathbf{k'^j} \\
f'_j &= f_i * A^i_j
\end{aligned}$$

Vi ser at også koordinatene til $\mathbf{e}$ transformerer kovariant, og indekserer derfor koordinatene til elementet med senket indeks.

3.7 Samlet spesifikasjon

Vi kan nå oppsummere bruddstykkene fotan til en fullstendig spesifikasjon som vi senere skal basere vår implementasjon på:

```
////////////////////////////////////////////////////
//                                                //
//                     RING                       //
//                                                //
////////////////////////////////////////////////////


specification RING
parameters
   sort        Ring
   operations
      0      :              -> Ring
      1      :              -> Ring
      - _    : Ring         -> Ring
      _ + _  : Ring * Ring  -> Ring
      _ * _  : Ring * Ring  -> Ring
specifies
   variables a, b, c : Ring
   equations
      a + 0          ==   a                       //  K1
      a + b          ==   b + a                   //  K2
      a + ( b + c )  ==   ( a + b ) + c           //  K3
      a + ( - a )    ==   0                       //  K4
      a * 1          ==   a                       //  K5
      a * ( b * c )  ==   ( a * b ) * c           //  K6
      ( b + c ) * a  ==   ( b * a ) + ( c * a )   //  K7
      a * ( b + c )  ==   ( a * b ) + ( a * c )   //  K8
end specification


////////////////////////////////////////////////////
//                                                //
//                     MODUL                      //
//                                                //
////////////////////////////////////////////////////

specification MODUL
include BOOL, HELTALL
parameters
   sort           Ring, Modul
   operations
      0              :                                -> Ring
      1              :                                -> Ring
      - _            : Ring                           -> Ring
      _ + _          : Ring * Ring                    -> Ring
```

```
      _ * _            : Ring * Ring                              -> Ring

      0                :                                          -> Modul
      _ + _            : Modul * Modul                            -> Modul
      - _              : Modul                                    -> Modul
      _ * _            : Ring * Modul                             -> Modul
      dimensjon ( _ )  : Modul                                    -> Heltall
      koordinat ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
      basis     ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1)) -> Modul
uses
      linkomb ( _,_ )  : Heltall * Modul | (0<=#<=dimensjon(#2)) -> Modul
specifies
   variables    m, n    : Ring
                a, b, c : Modul
                i, j    : Heltall
   equations
      RING ( Ring, 0, 1, -, +, * )
      a + 0           ==   a                  //   M1
      a + ( - a )     ==   0                  //   M2
      a + ( b + c )   ==   ( a + b ) + c      //   M3
      a + b           ==   b + a              //   M4
      m * ( a + b )   ==   m * a + m * b      //   M5
      ( m + n ) * a   ==   m * a + n * a      //   M6

                    ( 0 < dimensjon ( a ) )         == sann
                    dimensjon ( a )                 == dimensjon ( b )
                    basis ( a, i )                  == basis ( b , i )
                    linKomb ( dimensjon ( a ), a )  == a
      ( i  = j ) => koordinat ( basis ( a, i ) , j ) == 1
      ( i <> j ) => koordinat ( basis ( a, i ) , j ) == 0

      linKomb ( 1, a ) == koordinat ( a, 1 ) * basis ( a, 1 )
      linKomb ( i, a ) == koordinat ( a, i ) * basis ( a, i ) + linKomb ( i - 1, e )
end specification



//////////////////////////////////////////////////////
//                                                  //
//                 KOMODUL                          //
//                                                  //
//////////////////////////////////////////////////////

specification KOMODUL
include BOOL, HELTALL
parameters
   sort          Ring, Modul, koModul
   operations
      0                 :                                          -> Ring
      1                 :                                          -> Ring
```

```
    - _               : Ring                                       -> Ring
    _ + _             : Ring * Ring                                -> Ring
    _ * _             : Ring * Ring                                -> Ring

    0                 :                                            -> Modul
    _ + _             : Modul * Modul                              -> Modul
    - _               : Modul                                      -> Modul
    _ * _             : Ring * Modul                               -> Modul
    dimensjon ( _ )   : Modul                                      -> Heltall
    koordinat ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1))    -> Ring
    basis     ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1))    -> Modul

    0                 :                                            -> koModul
    _ + _             : koModul * koModul                          -> koModul
    - _               : koModul                                    -> koModul
    _ * _             : Ring * koModul                             -> koModul
    _ ( _ )           : koModul * Modul                            -> Ring
    dimensjon ( _ )   : koModul                                    -> Heltall
    koordinat ( _,_ ) : koModul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
    basis     ( _,_ ) : koModul * Heltall | (0<#2<=dimensjon(#1)) -> koModul
    _ ( _ )           : koModul * Modul                            -> Ring
specifies
   variables  m, n : Modul
              a, b : koModul
              k    : Ring
              i, j : Heltall
   equations
      RING     ( Ring, 0, 1, -, +, *    )
      MODUL    ( Ring, Modul,
                 0, 1, -, +, * ,
                 0, +, -, *, dimensjon, koordinat, basis   )
      MODUL    ( Ring, koModul,
                 0, 1, -, +, * ,
                 0, +, -, *, dimensjon, koordinat, basis   )

      a ( m + n )        ==   a ( m ) + a ( n )     \\   L1
      a ( k * m )        ==   k * a ( m )           \\   L2
      0 ( m     )        ==   0                     \\   L3
      ( a + b ) ( m )    ==   a ( m ) + b ( m )     \\   L4
      ( - a   ) ( m )    ==   - ( a ( m ) )         \\   L5
      ( k * a ) ( m )    ==   k * a ( m )           \\   L6

      ( i  = j ) =>  basis ( a, i ) ( basis ( m, j ) ) == 1
      ( i <> j ) =>  basis ( a, i ) ( basis ( m, j ) ) == 0
end specification
```

Chapter 4

Tensorer

Det finnes flere matematiske spesifikasjoner av tensorer, og ingen av dem er særlig passende for en direkte implementasjon i et programmeringsspråk. Vi skal i dette kapittelet først se på den matematiske definisjonen som vi bruker som utgangspunkt, og etterpå konstruere en algebraisk spesifikasjon som formaliserer definisjonen.

4.1 Matematisk bakgrunn

4.1.1 Kartesisk produkt

Gitt modulene $E_1, E_2, ..., E_k$ over samme ring R, der modul E_i har basismengde B_i. Vi definerer nå det *kartesiske produktet* F til å være:

$$F = E_1 \times \ldots \times E_k$$

der elementene i F består av k-tuplene $(\mathbf{v_1}, \ldots, \mathbf{v_k})$, for alle $\mathbf{v_i} \in V_i$. D.v.s.:

$$F = \{(\mathbf{v_1}, \ldots, \mathbf{v_k}) | \mathbf{v_i} \in V_i\}$$

□

Vi konstruerer operasjonene $\mathbf{0_F}$, '$-_F$', '$+_F$' og '$*_F$' på følgende måte:

$$\begin{array}{lcl}
\mathbf{0_F} & == & (\mathbf{0_1}, \ldots, \mathbf{0_k}) \\
(\mathbf{v_1}, \ldots, \mathbf{v_k}) +_F (\mathbf{w_1}, \ldots, \mathbf{w_k}) & == & (\mathbf{v_1} + \mathbf{w_1}, \ldots, \mathbf{v_k} + \mathbf{w_k}) \\
-_F(\mathbf{v_1}, \ldots, \mathbf{v_k}) & == & (-\mathbf{v_1}, \ldots, -\mathbf{v_k}) \\
c *_F (\mathbf{v_1}, \ldots, \mathbf{v_k}) & == & (c * \mathbf{v_1}, \ldots, c * \mathbf{v_k})
\end{array}$$

og det er nå lett å bevise at F er en modul over R.

4.1.2 Multilineære avbildninger

La $E_1, \ldots, E_k, F$ være moduler over ring K. En funksjon:

$$A : E_1 \times \ldots \times E_k \rightarrow F$$

er k-multilineær dersom A er lineær i alle moduler separat. Rommet av alle multilineære avbildninger fra $E_1 \times \ldots \times E_k$ til K benevner vi heretter $L(E_1 \times \ldots \times E_k; F)$.□

Et eksempel på dette er:

- $A((k * u_1, u_2, \ldots, u_k)) = k * A((u_1, u_2, \ldots, u_k))$
- $A((u_1, k * u_2, \ldots, u_k)) = k * A((u_1, u_2, \ldots, u_k))$
- $A((u_1 + u_1', u_2, \ldots, u_k)) = A((u_1, u_2, \ldots, u_k)) + A((u_1', u_2, \ldots, u_k))$
- $A((u_1, u_2 + u_2', \ldots, u_k)) = A((u_1, u_2, \ldots, u_k)) + A((u_1, u_2', \ldots, u_k))$

4.1.3 Kommutativitet

Vi har til nå ikke forutsatt at en ring R er kommutativ med hensyn på multiplikasjon. Det vil si at vi ikke har antatt at:

$$\forall a, b \in R : a * b = b * a$$

Det viser seg at vi ikke kan ha multilineære avbildninger fra moduler over ringer som ikke er kommutative. Grunnen til dette er at definisjonen av multilinearitet da er underbestemt. Det kan illustreres ved foelgende eksempel:

$$\begin{aligned} \mathbf{A}((k_1 * \mathbf{u_1}, k_2 * \mathbf{u_2}, \ldots, \mathbf{u_k})) &= k_1 * \mathbf{A}((\mathbf{u_1}, k_2 * \mathbf{u_2}, \ldots, \mathbf{u_k})) \\ &= k_1 * k_2 * \mathbf{A}((\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k})) \end{aligned}$$

Vi kan og utlede at:

$$\begin{aligned} \mathbf{A}((k_1 * \mathbf{u_1}, k_2 * \mathbf{u_2}, \ldots, \mathbf{u_k})) &= k_2 * A((k_1 * \mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k})) \\ &= k_2 * k_1 * A((\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k})) \end{aligned}$$

Dette betyr videre:

$$\begin{aligned} k_1 * k_2 * A((\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k})) &= k_2 * k_1 * A((\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k})) \\ &\Updownarrow \\ k_1 * k_2 &= k_2 * k_1 \end{aligned}$$

Dette betyr at vi alltid må kreve at ringen R er kommutativ når vi snakker om multilinearitet.

Multilineære avbildninger har en del nyttige egenskaper. La modulen E_i ha basis $B_i = \{\mathbf{b_{i,1}}, \ldots, \ldots, \mathbf{b_{i,d_i}}\}$. Da kan alle multilineære avbildninger $A \in L(E_1 \times \ldots \times E_k; K)$ skrives som en lineær kombinasjon:

$$\mathbf{A} = a_{i_1,\ldots,i_k} * (\mathbf{b^{1,i_1}}, \ldots, \mathbf{b^{k,d_k}})$$

for alle $a^{i_1,\ldots,i_k} \in K$ og $\{\mathbf{b^{i,j_1}}, \ldots, \mathbf{b^{i,j_i}}\}$ er basis for E_i^*. Dette vil være helt sentralt for valg av datastruktur i en implementasjon av tensorer.

4.1.4 Def. tensorer

Gitt en modul E over en ring K, og la E^* være den duale modul til E. Vi definerer mengden:

$$T_s^r(E) = L((E^*)^r \times E^s; K)$$

Elementene i mengden $T_s^r(E)$ kalles tensorer. La t være et element i $T_s^r(E)$. Vi sier at t er *kontravariant* i r argumenter og *kovariant* i s argumenter, siden siden t er en multilineær avbildning som tar det kartesiske produktet av r komodulelementer og s modulelement som argument. Vi sier videre at t er av type (r, s).□

En kan si at en tensor over en modul E og en ring K er en generalisering av de kartesiske produktene mellom modulen E og den duale modul E^*. Dette kommer klart frem fra en operasjon som kalles *tensorprodukt* mellom to tensorer. Et tensorprodukt kan lettest betraktes som en operasjon som, gitt to elementer fra det kartesiske produkt mellom E og E^* (d.v.s. to tensorer), danner et nytt element i det kartesiske produktet av argumentene.

Teorem

Gitt en ring K og en modul E over K. Da eksisterer det en *isomorfisme* mellom K og $T_0^0(E)$.□

Beviset er forholdsvis enkelt, og det holder derfor å bare skissere argumentasjonen. $T_0^0(E)$ er mengden av alle lineære avbildninger som ikke tar noen argumenter, og returnerer et element i K. D.v.s. at $T_0^0(E)$ er mengden av alle konstanter i K.

Teorem

Gitt en modul E over en ring K. Da eksisterer det en isomorfisme mellom E^*, og $T_1^0(E)$.□

Dette er rimelig, siden $T_1^0(E)$ er mengden av alle multilineære funksjoner som tar et element fra E som argument, og returnerer et element i K. Siden E^* er mengden av alle lineære funksjoner som tar et element fra E som argument, må $T_1^0(E)$ og E^* være identiske.

Teorem

Gitt en modul E over en ring K. Da eksisterer det en isomorfisme mellom E og $T_0^1(E)$.□

Argumentasjonen følger samme form som forrige teorem. Matematisk kan denne isomorfismen uttrykkes som '$\mathbf{v} \mapsto$ _ ($\mathbf{v}$)' for alle $\mathbf{v} \in E$.

4.2 Algebraisk spesifikasjon

4.2.1 Modulegenskaper

Vi vet at mengden av alle lineære avbildninger fra en modul M over en ring R kan gis modulegenskaper. Siden multilineære avbildninger også er lineære avbildninger, må dette bety at hver 'skive'av $T_s^r(E)$ kan gis modulegenskaper over ringen R. Med skive mener vi alle tensorer av samme type (r, s). Vi vet derfor at en skive $T_s^r(E)$ kan ha følgende operasjoner:

```
0       :                   -> Tensor
_ + _   : Tensor * Tensor   -> Tensor
- _     : Tensor            -> Tensor
_ * _   : Ring * Tensor     -> Tensor
```

der sorten *Tensor* har type (r, s). Sammen med ringen *Ring* og alle tilhørende ringoperasjonene må sorten *Tensor* tilfredstille spesifikasjonen MODUL. Siden det kun er tensorer av samme type som oppfyller modulegenskapene, må det bety at alle forskjellige typer av tensorer har sine egne nullelementer. For å kunne generere disse nullelementene utvider vi signaturen til operasjonen '0':

```
0 ( _ )  : Tensor -> Tensor
```

Denne operasjonen vil generere et nullelement av samme type som argumentet til funksjonen. Det er en nær sammenheng mellom tensorer av forskjellige typer. Dette skal vi utnytte ved å gi en spesifikasjon som dekker tensorer av alle typer. Vi må derfor utvide spesifikasjonen med observatørene:

```
antKomoduler ( _ )   : Tensor            -> Heltall
antModuler   ( _ )   : Tensor            -> Heltall
sammeType    ( _,_ ) : Tensor * Tensor   -> Bool
```

der operasjonen *antKomoduler (t)* returnerer antallet komodulelementer t tar som argument, og operasjonen *antmoduler (t)* returnerer antallet modulelementer t trenger som argument. Operasjonen *sammeType (t1,t2)* er en sammensatt operasjon som returnerer *sann* hvis $t1$ og $t2$ er av samme type, *usann* ellers. Vi kan sette opp følgende aksiomer for operasjonene:

```
variables   t1, t2 : Tensor
   sammeType ( t1, t2 ) == (antKomoduler(t1)   =  antKomoduler(t2) /\
                            antModuler  (t1)   =  antModuler  (t2)   )
equations
   MODUL ( Ring,
           Tensor | sammeType( #1, t1 ),           //   NB !
           0, 1, -, +, * ,
           0 ( t1 ),                               //   NB !
           +, -, *      )
```

Merk at vi påstår modulegenskaper kun for tensorer av samme type. Det oppnår vi ved å parametrisere spesifikasjonen med sorten *'Tensor | sammeType(#1,t1)'*. Denne sorten består av alle tensorer i sorten *Tensor* som oppfyller predikatet *sammeType(#1,t1)*. Dette fører til at alle tensorer som ikke tilhører samme skive som tensoren $t1$ blir 'filtrert' ut av sorten *Tensor*. Siden tensoren $t1$ er en allkvantorisert variabel og dermed spenner over tensorer av alle typer, får vi påstatt at modulegenskapene skal gjelde for hver skive av sorten *Tensor*. Merk og at vi ikke kan bruke samme nullelement for tensorer av alle typer. Vi bruker heller null-generatoren parametrisert med tensoren $t1$.

Vi må og presisere at moduloperasjonene gir nye tensorelementer som er av samme type som $t1$. Det oppnår vi ved å legge til følgende aksiomer:

```
variables   t1, t2 : Tensor
            k      : Ring
equations
   antKomoduler ( 0 ( t1 ) ) == antKomoduler ( t1      )
   antModuler   ( 0 ( t1 ) ) == antModuler   ( t1      )
   antKomoduler (     - t1 ) == antKomoduler ( t1      )
   antModuler   (     - t1 ) == antModuler   ( t1      )
   antKomoduler (  t1 + t2 ) == antKomoduler ( t1 + t2 )
   antModuler   (  t1 + t2 ) == antModuler   ( t1 + t2 )
   antKomoduler (   a * t1 ) == antKomoduler ( t1      )
   antModuler   (   a * t1 ) == antModuler   ( t1      )
```

4.2.2 Isomorfismer

Vi vet fra teoremene foran at det er flere isomorfismer mellom forskjellige moduler og tensorer. Vi kan spesifisere isomorfismene ved å utstyre sorten med følgende 'konverterings-operatorer':

```
operations
   //   Generatorer
   tensor     ( _ ) : Ring                        -> Tensor
```

```
tensor    ( _ ) : Modul                     -> Tensor
tensor    ( _ ) : Komodul                   -> Tensor
//  Observatorer
ring      ( _ ) : Tensor | erRing    (#1)   -> Ring
modul     ( _ ) : Tensor | erModul   (#1)   -> Modul
komodul   ( _ ) : Tensor | erKomodul(#1)    -> Komodul
//  Predikater
erRing    ( _ ) : Tensor                    -> Bool
erModul   ( _ ) : Tensor                    -> Bool
erKomodul ( _ ) : Tensor                    -> Bool
```

Her må vi kreve at sorten *Komodul* tilfredstiller spesifikasjonen KOMODUL sammen med sorten *Modul* og *Ring*. Vi kjenner også typen til tensorene som blir generert, og vi får derfor følgende aksiomer:

```
variables   m : Modul
            v : Komodul
            k : Ring
equations
      KOMODUL     ( Ring, Modul, Komodul,
                    0, 1, -, +, * ,
                    0, +, -, * ,
                    0, +, -, *        )
      //  Isomorfismer
      ring     ( tensor ( a ) ) == a
      modul    ( tensor ( m ) ) == m
      komodul  ( tensor ( v ) ) == v

      //  Spesifiserer type
      antKomoduler ( tensor ( a ) ) == 0
      antModuler   ( tensor ( a ) ) == 0
      antKomoduler ( tensor ( m ) ) == 1
      antModuler   ( tensor ( m ) ) == 0
      antKomoduler ( tensor ( v ) ) == 0
      antModuler   ( tensor ( v ) ) == 1
      //  Predikater
      erRing    ( t1 )    == (antKomoduler(t1)=0 /\ antModuler ( t1 )=0)
      erModul   ( t1 )    == (antKomoduler(t1)=1 /\ antModuler ( t1 )=0)
      erKomodul ( t1 )    == (antKomoduler(t1)=0 /\ antModuler ( t1 )=1)
```

Disse isomorfismene er helt grunnleggende for oss, siden tensorene som her blir generert vil danne grunnlaget for tensorer av alle andre typer.

4.2.3 Tensorprodukt og Indreprodukt

Til nå har vi bare spesifisert generatoroperasjoner som konstruerer tensorer av type $(0,0)$, $(0,1)$ og $(1,0)$. Tensorene av disse typene er observerbare på grunn av observatørene *ring*, *modul* og *komodul*. For å generere tensorer av nye typer trenger vi derfor en ny generator. Til dette formålet har vi operasjonen *tensorprodukt*. Et tensorprodukt mellom to tensorer kan betraktes som en funksjon som genererer en ny tensor av en type som er mer 'utilgjengelig' med hensyn på observasjon enn argumentene. Vi trenger derfor også en operasjon for å få resultatet av et tensorprodukt 'ned' mot de observerbare tensortypene. Denne operasjonen kalles ofte *indreprodukt*. Uformelt kan en betrakte en applikasjon av operasjonen *indreprodukt* som en forsinket evaluering av et 'uttrykk' som operasjonen *tensorprodukt* har bygget opp av sine argumenter. Vi utvider derfor operatormengden til sorten *Tensor* med følgende to operasjoner:

```
_ * _   : Tensor * Tensor -> Tensor          \\  Tensorprodukt
< _,_ > : Tensor * Tensor -> Tensor          \\  Indreprodukt
```

Vi må nå spesifisere oppførselen til disse operasjonene. Operasjonene er gjensidig avhengige, og vi kan derfor ikke spesifisere dem hver for seg. Vi kan likevel spesifisere hver av operasjonene for de tensorene vi kan observere:

```
variables
      a, b                 : Ring
      m, n                 : Modul
      v, w                 : Komodul
      t1, t2, t3, t4, t5   : Tensor
equations
      //  indreprodukt
      < tensor ( a ), tensor ( b ) > ==  tensor (   a * b    )
      < tensor ( w ), tensor ( m ) > ==  tensor (   w ( m ) )
      < tensor ( m ), tensor ( w ) > ==  tensor (   w ( m ) )

      //  tensorprodukt
      tensor ( a ) * tensor ( b )    ==  tensor ( a * b )
      tensor ( a ) * tensor ( m )    ==  tensor ( a * m )
      tensor ( a ) * tensor ( w )    ==  tensor ( a * w )
```

Vi må og spesifisere hvordan *tensorprodukt* og *indreprodukt* er relatert for tensorer av alle andre typer enn basistilfellene. Vi må derfor spesifisere sammenhengen mellom operasjonene for alle andre uttrykk av typen *Tensor*. Det er likevel en del kombinasjoner av argumenter som ikke er definert, og vi vil derfor kun ha aksiomene:

```
< < t1,  tensor( a ) >, t2 >        == < t1, tensor ( a ) * t2 >
<   t1 * tensor( a )   , t2 >       == < t1, tensor ( a ) * t2 >
```

```
< t1 * tensor( m ), tensor ( w ) > == t1 * < tensor ( m ), tensor ( w ) >
< t1 * tensor( v ), tensor ( n ) > == t1 * < tensor ( v ), tensor ( n ) >
< t1 * tensor( m ), tensor ( n ) > == < t1, tensor ( n ) > * tensor ( m )
< t1 * tensor( v ), tensor ( w ) > == < t1, tensor ( w ) > * tensor ( v )
< t1 , t2 * t3 >                   == < < t1, t3 > , t2 >
```

Et typisk uttrykk som ikke er gyldig er '$<$ *tensor(m), tensor(n)* $>$'. Dette kan betraktes som en typefeil, siden *tensor(m)* er en funksjon som tar en tensor av type $(1,0)$ som argument mens *tensor(n)* er en tensor av type $(0,1)$. Operasjonen *indreprodukt* er derfor partiell. Det kan fra definisjonen over utledes at følgende predikat er en gyldig vokter for operasjonen:

```
operation
   gyldigIndreprod ( _, _ ) : Tensor * Tensor -> Bool
equation
   gyldigIndreprod( t1,t2 ) == ( antModuler  (t2) <= antKomoduler(t1) /\
                                 ( antKomoduler(t2) <= antModuler  (t1)    )
```

Vi må også forsikre oss om at *indreprodukt* er multilineær i alle argumenter separat. Med dette mener vi at alle multilineære kombinasjoner av *tensorprodukt* skal distribueres ut av operasjonen *indreprodukt*:

```
//   linearitet
< t1 + t2, t3 >  == < t1, t3 > + < t2, t3 >
< t1, t2 + t3 >  == < t1, t2 > + < t1, t3 >
< a * t1, t2  >  == a * < t1, t2 >
< t1, a * t2  >  == a * < t1, t2 >

//   multilinearitet
< t1, ( t2 + t3 ) * t4 > == < t1, t2 * t4 > + < t1, t3 * t4 >
< t1, t2 * ( t3 + t4 ) > == < t1, t2 * t3 > + < t1, t2 * t4 >
< t1, ( a * t2 ) * t3  > == a * < t1, t2 * t3 >                 //   NB !
< t1, t2 * ( a * t3 )  > == a * < t1, t2 * t3 >                 //   NB !
```

Merk at de to siste likningene krever at ringen er kommutativ med hensyn på multiplikasjon. Hvis ringen ikke er kommutativ vil likningene over føre til *forvirring* i sorten *Ring*.

Operasjonene *indreprodukt* og *tensorprodukt* lager tensorer av nye typer. Typen til disse tensorene må vi kunne observere, og vi kan utlede følgende aksiomer:

```
antKomoduler ( t1 * t2  ) ==  antKomoduler(t1) + antKomoduler(t2)
antModuler   ( t1 * t2  ) ==  antModuler  (t1) + antModuler  (t2)
antKomoduler ( < t1,t2> ) ==  antKomoduler(t1) - antModuler  (t2)
antModuler   ( < t1,t2> ) ==  antModuler  (t1) - antKomoduler(t2)
```

4.2.4 Kontraksjon

En operasjon som er mye brukt på tensorer er *kontraksjon*. Denne operasjonen kan betraktes som et forsinket indreprodukt på tensorer. Det kan lettest vises ved følgende definisjon:

```
kontraksjon ( t1*tensor(m) ) == < t1, tensor ( m ) >
kontraksjon ( t1*tensor(v) ) == < t1, tensor ( v ) >
```

Siden operasjonen er en forsinket applikasjon, kan operasjonen kun ta som argument en tensor som kan 'evalueres'. Denne operasjonen er derfor partiell, og vi benytter predikatet *kanKontraktere* som vokter. Spesifikasjonen blir defor:

```
operation
   kontraksjon     ( _ ) : Tensor | kanKontraktere ( #1 ) -> Tensor
   kanKontraktere ( _ ) : Tensor                          -> Bool
variables   t1 : Tensor
            m  : Modul
            v  : Komodul
equation
   antKomoduler ( kontraksjon ( t1 ) ) == antKomoduler ( t ) - 1
   antModuler   ( kontraksjon ( t1 ) ) == antModuler   ( t ) - 1
   kanKontraktere ( t1 ) == 0 < antModuler(t1) /\ 0 < antKomoduler(t1)
   kontraksjon ( t1 * tensor(m) ) == < t1, tensor ( m ) >
   kontraksjon ( t1 * tensor(v) ) == < t1, tensor ( v ) >
```

4.2.5 Kronecker tensor

Det finnes en spesiell tensor t av type (1, 1) som har egenskapen:

```
< t, tensor ( v ) * tensor ( m ) > == tensor ( v ( m ) )
```

Denne tensoren er mye brukt innenfor tensorberegninger, og refereres til som *Kronecker tensor*. Siden spesifikasjonen kun sier at denne tensorer eksisterer, må vi ha en egen generator for tensoren. Vi må derfor utvide spesifikasjonen for tensorer med følgende:

```
operation
  Kronecker  :  -> Tensor
equations
   antModuler   ( Kronecker )                         == 1
   antKomoduler ( Kronecker )                         == 1
   < Kronecker, ( tensor ( v ) * tensor ( m ) > == tensor ( v ( m ) )
```

4.3 Algoritmer og datastruktur ved implementasjon

Gitt en modul M over en ring R der M har basis $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$. Vi har og en komodul M^* med dual basis $B' = \{\mathbf{b'^1}, \ldots, \mathbf{b'^d}\}$. Vi vet at alle tensorer $\mathbf{t}$ av type (r, s) kan skrives som en lineær kombinasjon:

$$t = t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} * (\mathbf{b_{i_1}}, \ldots, \mathbf{b_{i_r}}, \mathbf{b'^{j_1}}, \ldots, \mathbf{b'^{j_s}})$$

der $0 < i_1, \ldots, i_r, j_1, \ldots, j_s \leq d$ og $t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} \in R$. Siden vi alltid kan konstruere basiselementene $(\mathbf{b_{i_1}}, \ldots, \mathbf{b_{i_r}}, \mathbf{b'^{j_1}}, \ldots, \mathbf{b'^{j_s}})$ til en tensor av type (r, s) hvis vi har tilgang til elementene i B og B', trenger vi ikke lagre basiselementene i noen datastruktur, vi trenger kun lagre koordinatene til t. Vi kan lett finne at t inneholder d^{r+s} koordinater, og at dette er dimensjonen til modulen $T^r_s(E)$. Siden $T^r_s(E)$ kan gis modulegenskaper kan vi derfor benytte følgende algoritmer for å implementere moduloperasjonene '0', '+', '-' og '*':

$$\begin{array}{lcll}
\forall t \in T^r_s(E) & : & (0(t))^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} & = & 0 \\
\forall t_1, t_2 \in T^r_s(E) & : & (t_1 + t_2)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} & = & (t_1)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} + (t_2)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} \\
\forall t \in T^r_s(E) & : & (-t)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} & = & -(t)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} \\
\forall k \in R, \forall t \in T^r_s(E) & : & (k * t)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} & = & k * (t)^{i_1,\ldots,i_r}_{j_1,\ldots,j_s}
\end{array}$$

summert over alle $i_1, \ldots, i_r, j_1, \ldots, j_s$.

4.3.1 Konverteringsoperatorer

I matematisk notasjon uttrykkes tensorer og moduler på samme form. Det fører til mye forvirring p.g.a. mye implisitt typekonvertering. Vi vet at et element $\mathbf{m} \in M$ skrives som den lineære kombinasjonen:

$$m = m^i * \mathbf{b_i}$$

Vi skal videre representere tensorer av type $(0,0)$, $(0,1)$ og $(1,0)$ på følgende form:

$$\begin{array}{lcll}
k \in R & \Rightarrow & tensor(k) & = & k * () \\
m \in M & \Rightarrow & tensor(m) & = & m^i * (\mathbf{b_i}) \\
v \in M^* & \Rightarrow & tensor(v) & = & v_i * (\mathbf{b'^i})
\end{array}$$

Dette betyr at mengden $\{()\}$ er basismengde for alle tensorer av type $(0,0)$, mengden $\{(\mathbf{b_1}), \ldots, (\mathbf{b_d})\}$ er basismengde for tensorer av type $(1,0)$, og mengden $\{(\mathbf{b'^1}), \ldots, (\mathbf{b'^d})\}$ er basismengde for tensorer av type $(0,1)$. Hvis modulene M og M^* har generatorer som returnerer basiselementene til modulene kan vi derfor lett konvertere tensorer til moduler. Konverteringen fra tensor til et ringelement er triviell.

```
      0 ( _ )           : Tensor                                  -> Tensor
      _ + _             : Tensor * Tensor                         -> Tensor
      - _               : Tensor                                  -> Tensor
      _ * _             : Ring * Tensor                           -> Tensor
      dimensjon ( _ )   : Tensor                                  -> Heltall
      koordinat ( _,_ ) : Tensor * Heltall | (0<#2<=dimensjon(#1)) -> Ring
      basis     ( _,_ ) : Tensor * Heltall | (0<#2<=dimensjon(#1)) -> Tensor

      _ * _             : Tensor * Tensor                         -> Tensor
      < _,_ >           : Tensor * Tensor | gyldigIndreprod(#1,#2) -> Tensor
      kontraksjon ( _ ) : Tensor          | kanKontraktere (#1)   -> Tensor
      Kronecker         :                                         -> Tensor

      //   sammensatte operasjoner
      erRing          ( _ )    : Tensor                           -> Bool
      erModul         ( _ )    : Tensor                           -> Bool
      erKomodul       ( _ )    : Tensor                           -> Bool
      sammeType       ( _,_ )  : Tensor * Tensor                  -> Bool
      gyldigIndreprod ( _, _ ) : Tensor * Tensor                  -> Bool
      kanKontraktere  ( _ )    : Tensor                           -> Bool
specifies
   variables  a, b, k              : Ring
              m, n                 : Modul
              v, w                 : Komodul
              t1, t2, t3, t4, t5   : Tensor
   equations
      KOMODUL    ( Ring, Modul, koModul,
                   0, 1, -, +, * ,
                   0, +, -, * ,dimensjon, koordinat, basis ,
                   0, +, -, * ,dimensjon, koordinat, basis     )
      MODUL      ( Ring,
                   Tensor | sammeType( #1, t1 ),          //   NB !
                   0, 1, -, +, * ,
                   0 ( t1 ),                              //   NB !
                   +, -, *, dimensjon, koordinat, basis )

      ring        ( tensor ( a )  ) == a
      modul       ( tensor ( m )  ) == m
      komodul     ( tensor ( v )  ) == v

      antKomoduler ( tensor ( a )  ) ==  0
      antModuler   ( tensor ( a )  ) ==  0
      antKomoduler ( tensor ( m )  ) ==  1
      antModuler   ( tensor ( m )  ) ==  0
      antKomoduler ( tensor ( v )  ) ==  0
      antModuler   ( tensor ( v )  ) ==  1
      antModuler   ( Kronecker     ) ==  1
      antKomoduler ( Kronecker     ) ==  1
      antKomoduler ( 0 ( t1 )      ) ==  antKomoduler ( t       )
      antModuler   ( 0 ( t1 )      ) ==  antModuler   ( t       )
```

```
    antKomoduler (      - t      ) ==  antKomoduler ( t       )
    antModuler   (      - t      ) ==  antModuler   ( t       )
    antKomoduler (  t1 + t2      ) ==  antKomoduler ( t1 + t2 )
    antModuler   (  t1 + t2      ) ==  antModuler   ( t1 + t2 )
    antKomoduler (   a * t1      ) ==  antKomoduler ( t1      )
    antModuler   (   a * t1      ) ==  antModuler   ( t1      )
    antKomoduler ( t1 * t2       ) ==  antKomoduler(t1) + antKomoduler(t2)
    antModuler   ( t1 * t2       ) ==  antModuler  (t1) + antModuler  (t2)
    antKomoduler ( < t1,t2>      ) ==  antKomoduler(t1) - antModuler  (t2)
    antModuler   ( < t1,t2>      ) ==  antModuler  (t1) - antKomoduler(t2)

    antKomoduler ( kontraksjon ( t1 ) )   == antKomoduler ( t ) - 1
    antModuler   ( kontraksjon ( t1 ) )   == antModuler   ( t ) - 1

    < tensor ( a ), tensor ( b ) >        ==  tensor (   a * b   )
    < tensor ( w ), tensor ( m ) >        ==  tensor (   w ( m ) )
    < tensor ( m ), tensor ( w ) >        ==  tensor (   w ( m ) )
    tensor ( a ) * tensor ( b )           ==  tensor ( a * b )
    tensor ( a ) * tensor ( m )           ==  tensor ( a * m )
    tensor ( a ) * tensor ( w )           ==  tensor ( a * w )

    < < t1,  tensor ( a ) >, t2 >         ==  < t1, tensor ( a ) * t2 >
    <   t1 * tensor ( a )   , t2 >        ==  < t1, tensor ( a ) * t2 >
    < t1 * tensor ( m ) , tensor ( w ) >  ==  t1 * < tensor ( m ), tensor ( w ) >
    < t1 * tensor ( v ) , tensor ( n ) >  ==  t1 * < tensor ( v ), tensor ( n ) >
    < t1 * tensor ( m ) , tensor ( n ) >  ==  < t1, tensor ( n ) > * tensor ( m )
    < t1 * tensor ( v ) , tensor ( w ) >  ==  < t1, tensor ( w ) > * tensor ( v )
    < t1 , t2 * t3 >                      ==  < < t1, t3 > , t2 >
    < t1 + t2, t3 >                       ==  < t1, t3 > + < t2, t3 >
    < t1, t2 + t3 >                       ==  < t1, t2 > + < t1, t3 >
    < a * t1, t2  >                       ==  a * < t1, t2 >
    < t1, a * t2  >                       ==  a * < t1, t2 >
    < t1, ( t2 + t3 ) * t4 >              ==  < t1, t2 * t4 > + < t1, t3 * t4 >
    < t1, t2 * ( t3 + t4 ) >              ==  < t1, t2 * t3 > + < t1, t2 * t4 >
    < t1, ( a * t2 ) * t3  >              ==  a * < t1, t2 * t3 >
    < t1, t2 * ( a * t3 )  >              ==  a * < t1, t2 * t3 >

    kontraksjon ( t1 * tensor(m) )        ==  < t1, tensor ( m ) >
    kontraksjon ( t1 * tensor(v) )        ==  < t1, tensor ( v ) >
    < Kronecker,( tensor (v)*tensor (m) > ==  tensor ( v ( m ) )


    erRing    ( t1 )          == ( antKomoduler(t1)=0 /\ antModuler ( t1 )=0)
    erModul   ( t1 )          == ( antKomoduler(t1)=1 /\ antModuler ( t1 )=0)
    erKomodul ( t1 )          == ( antKomoduler(t1)=0 /\ antModuler ( t1 )=1)
    sammeType ( t1, t2 )      == ( antKomoduler(t1)   =  antKomoduler(t2) /\
                                   antModuler  (t1)   =  antModuler  (t2)   )
    gyldigIndreprod( t1,t2 ) == ( antModuler  (t2)   <= antKomoduler(t1) /\
                                 ( antKomoduler(t2)   <= antModuler  (t1)    )
    kanKontraktere ( t1 )     == 0 < antModuler(t1)   /\ 0 < antKomoduler(t1)
end specification
```

4.3.2 Tensorprodukt

Vi vet fra den algebraiske spesifikasjonen at tensorproduktet mellom to tensorer $t_1 \in T_{s1}^{r1}(E)$ og $t_2 \in T_{s2}^{r2}(E)$ er en tensor $(t_1 * t_2) \in T_{s1+s2}^{r1+r2}(E)$. Vi kan utlede at koordinatene til tensorproduktet $(t_1 * t_2)$ kan beregnes etter følgende algoritme:

$$(t_1 * t_2)_{j_1,\ldots,j_{s_1},n_1,\ldots,n_{s_2}}^{i_1,\ldots,i_{r_1},m_1,\ldots,m_{r_2}} = (t_1)_{j_1,\ldots,j_{s_1}}^{i_1,\ldots,i_{r_1}} * (t_2)_{n_1,\ldots,n_{s_2}}^{m_1,\ldots,m_{r_2}}$$

Basisen til $t_1 * t_2$ blir som definert tidligere for en tensor av $type(r_1 + r_2, s_1 + s_2)$. Vi kjenner igjen denne algoritmen som den vanlige definisjon for *tensorprodukt* i en del lærebøker (f.eks. [4]).

4.3.3 Indreprodukt

Elementene i et tensorrom $T_s^r(E)$ er pr. definisjon multilineære avbildninger fra det kartesiske produktet av r elementer fra E^* og s elementer fra E inn i ringen R. Et *indreprodukt* mellom to tensorer kan betraktes som en applikasjon av den ene tensoren på den andre, der resultatet ikke nødvendigvis er et element i R, men heller element i et tensorrom $T_{s'}^{r'}(E)$. Gitt to tensorer $t_1 \in T_{s_1}^{r_1}(E)$ og $t_2 \in T_{s_2}^{r_2}(E)$ vil indreproduktet $< t_1, t_2 >$ være et element i $T_{s_1-r_2}^{r_1-s_2}(E)$, og det kan vises at koordinatene til $< t_1, t_2 >$ kan beregnes på følgende måte:

$$< t_1, t_2 >_{j_1,\ldots,j_n}^{i_1,\ldots,i_m} = (t_1)_{j_1,\ldots,j_n,l_1,\ldots,l_{r_2}}^{i_1,\ldots,i_m,k_1,\ldots,k_{s_2}} * (t_2)_{k_1,\ldots,k_{s_2}}^{l_1,\ldots,l_{r_2}}$$

4.3.4 Kontraksjon

Kontraksjon er en operasjon på tensorer som kan betraktes som en forsinket applikasjon av en tensor på et argument. Det kan vises at koordinatene til resultatet av en kontraksjon av en tensor $t \in T_s^r(E)$, $(0 < r, s)$ er:

$$(kontraksjon(t))_{j_1,\ldots,j_{s-1}}^{i_1,\ldots,i_{r-1}} = t_{j_1,\ldots,j_{s-1},m}^{i_1,\ldots,i_{r-1},m}$$

summert over alle m. Vi vet og at kontraksjonen av t er av type $(r-1, s-1)$.

4.3.5 Kronecker tensor

Vi vet at tensoren *Kronecker* er av type $(1,1)$. Det er lett å vise at tensoren t som skrives med den lineære kombinasjonen:

$$Kronecker = \delta_j^i * (\mathbf{b_i}, \mathbf{b'^j})$$

der:

$$\delta_j^i = \begin{cases} 0 & \text{if } i <> j \\ 1 & \text{if } i = j \end{cases}$$

oppfyller den algebraiske spesifikasjonen for tensoren *Kronecker*. Koordinatene til tensoren *Kronecker* blir derfor $\{\delta_j^i\}$.

4.4 Samlet spesifikasjon

Vi kan samle spesifikasjonen for tensorer til foelgende:

```
//////////////////////////////////////////////////
//                                              //
//                   TENSOR                     //
//                                              //
//////////////////////////////////////////////////


specification TENSOR
include BOOL, HELTALL
parameters
   sort          Ring, Modul, koModul, Tensor
   operations
      0                 :                                           -> Ring
      1                 :                                           -> Ring
      - _               : Ring                                      -> Ring
      _ + _             : Ring * Ring                               -> Ring
      _ * _             : Ring * Ring                               -> Ring

      0                 :                                           -> Modul
      _ + _             : Modul * Modul                             -> Modul
      - _               : Modul                                     -> Modul
      _ * _             : Ring * Modul                              -> Modul
      dimensjon ( _ )   : Modul                                     -> Heltall
      koordinat ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1))   -> Ring
      basis     ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1))   -> Modul

      0                 :                                           -> koModul
      _ + _             : koModul * koModul                         -> koModul
      - _               : koModul                                   -> koModul
      _ * _             : Ring * koModul                            -> koModul
      _ ( _ )           : koModul * Modul                           -> Ring
      dimensjon ( _ )   : koModul                                   -> Heltall
      koordinat ( _,_ ) : koModul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
      basis     ( _,_ ) : koModul * Heltall | (0<#2<=dimensjon(#1)) -> koModul
      _ ( _ )           : koModul * Modul                           -> Ring

      //   observatorer
      ring         ( _ ) : Tensor | erRing   (#1)                   -> Ring
      modul        ( _ ) : Tensor | erModul  (#1)                   -> Modul
      komodul      ( _ ) : Tensor | erKomodul(#1)                   -> koModul
      antKomoduler ( _ ) : Tensor                                   -> Heltall
      antModuler   ( _ ) : Tensor                                   -> Heltall

      //   generatorer
      tensor ( _ )       : Ring                                     -> Tensor
      tensor ( _ )       : Modul                                    -> Tensor
      tensor ( _ )       : koModul                                  -> Tensor
```

Chapter 5

Derivasjonsoperatorer

I dette kapittelet skal vi se på hva derivasjonsoperatorer er, og vi skal utlede noen viktige egenskaper ved dem. Dette er viktig for oss å vite når vi senere skal spesifisere og implementere derivasjonsoperatorene *Lie derivasjon* og *partiell derivasjon.* Vi vil i dette kapittelet finne egenskaper som gjelder alle mulige derivasjonsoperatorer, ikke bare de operatorene vi skal jobbe videre med senere. Det som er særlig viktig å merke seg i dette kapittelet er at hvis vi har gitt en derivasjonsoperator D_r for en ring R og en derivasjonsoperator D_m for en modul M over R, kan vi finne en tilhørende derivasjonsoperator D_t for alle tensorer $\mathbf{t} \in T^r_s(E)$. Vi vil i dette kapittelet også finne en generell algoritme for å beregne resultatet av anvendelsen av D_t på $\mathbf{t}$. Denne algoritmen vil bli svært viktig for oss senere.

5.1 Ringderivasjon

En derivasjonsoperator D_r på en ring R er et lineært kart $D_r : R \to R$, som oppfyller følgende egenskaper:

1. $\forall x, y \in R : D_r(x + y) = D_r(x) + D_r(y)$
2. $\forall x, y \in R : D_r(x * y) = D_r(x) * y + x * D_r(y)$

En derivasjonsoperator D_r er triviell hvis og bare hvis for $\forall x \in R : D_r(x) = 0.\square$

Vi kan lett utlede følgende egenskaper for en derivasjonsoperator D_r på en ring R:

- $D_r(0) = 0$
- $D_r(1) = 0$
- $D_r(-x) = -D_r(x)$

Bevis :

$$\begin{aligned}
D_r(0) &= D_r(x * 0) \\
D_r(0) &= D_r(x) * 0 + x * D_r(0) \\
D_r(0) &= 0 + x * D_r(0) \\
D_r(0) &= x * D_r(0) \\
D_r(0) &= 0
\end{aligned}$$

og :

$$\begin{aligned}
D_r(1) &= D_r(1 * 1) \\
D_r(1) &= D_r(1) * 1 + 1 * D_r(1) \\
D_r(1) &= D_r(1) + D_r(1) \\
D_r(1) &= 0
\end{aligned}$$

og :

$$D_r(-x) = D_r((-1) * x)$$
$$D_r(-x) = D_r(-1) * x + (-1) * D_r(x)$$
$$D_r(-x) = 0 * x - D_r(x)$$
$$D_r(-x) = -D_r(x)$$

Eksempel :

La K være mengden av alle funksjoner $f : R \rightarrow R$ (der R er de reelle tall). D.v.s.:

$$K = \{f | f : R \rightarrow R\}$$

Vi kan nå definere K til å være en ring på følgende måte:

- $\forall x \in R : 0(x) = 0$
- $\forall x \in R : 1(x) = 1$
- $\forall x \in R, \forall a \in K : (-a)(x) = -a(x)$
- $\forall a, b \in K, \forall x \in R : (a + b)(x) = a(x) + b(x)$
- $\forall a, b \in K, \forall x \in R : (a * b)(x) = a(x) * b(x)$

I tillegg har vi derivasjonsoperatoren $D_r = \frac{d}{dx} : K \rightarrow K$, som er den 'normale' derivasjons-operatoren for en funksjon $f : R \rightarrow R$. Vi vet at $\frac{d}{dx}$ har følgende egenskaper :

- $\frac{d}{dx}(f + g) = \frac{d}{dx}(f) + \frac{\partial}{\partial x}(g)$
- $\frac{d}{dx}(f * g) = \frac{d}{dx}(f) * g + f * \frac{d}{dx}(g)$

Det betyr at $\frac{d}{dx}$ er en gyldig derivasjonsoperator for ringen K.

5.2 Modulderivasjon

En derivasjonsoperator D_m på en modul M over en ring R, er et lineært kart $D_m : M \to M$, som oppfyller følgende krav:

1. $\forall \mathbf{x}, \mathbf{y} \in M : D_m(\mathbf{x} + \mathbf{y}) = D_m(\mathbf{x}) + D_m(\mathbf{y})$
2. $\forall x \in R, \forall \mathbf{y} \in M : D_m(x * \mathbf{y}) = D_r(x) * \mathbf{y} + x * D_m(\mathbf{y})$

der D_r er en derivasjonsoperator for ringen R. En modulderivasjon D_m er triviell hvis og bare hvis $\forall \mathbf{x} \in M : D_m(\mathbf{x}) = \mathbf{0}$.□

La D_m være en derivasjonsoperator for modulen M. Siden $D_m(\mathbf{x})$ er et element i mengden M for alle $\mathbf{x} \in M$, vet vi at $D_m(\mathbf{x})$ kan skrives som en lineær kombinasjon av basisen $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ til M. Dette betyr at $D_m(\mathbf{b_i})$ kan uttrykkes som en lineær kombinasjon. La derfor koordinatmengden $E = \{e_1^1, \ldots, e_d^d\}$ være slik at følgende er oppfyllt:

$$D_m(\mathbf{b_i}) = e_i^j * \mathbf{b_j}$$

Hvis vi har gitt en derivasjonsoperator D_m for M, vet vi at mengden E alltid kan konstrueres. Dette vil vi ha nytte av når vi senere skal definere derivasjonsoperatorer for M sin duale modul.

5.3 Komodulderivasjon

En derivasjonsoperator D_k på en komodul M^* over en ring R er et lineært kart $D_k : M^* \to M^*$, som har følgende egenskaper:

$$\begin{array}{lclcl}
\forall \mathbf{x}, \mathbf{y} \in M^* & : & D_k(\mathbf{x} + \mathbf{y}) & = & D_k(\mathbf{x}) + D_k(\mathbf{y}) \\
\forall x \in R, \forall \mathbf{y} \in M^* & : & D_k(x * \mathbf{y}) & = & D_r(x) * \mathbf{y} + x * D_k(\mathbf{y}) \\
\forall \mathbf{y} \in M, \forall \mathbf{z} \in M^* & : & D_r(\mathbf{z}(\mathbf{y})) & = & (D_k(\mathbf{z}))(\mathbf{y}) + \mathbf{z}(D_m(\mathbf{y}))
\end{array}$$

der D_r er en derivasjonsoperator for ringen R, og D_m er en derivasjonsoperator for modulen M. Derivasjonsoperatoren D_k er triviell hvis og bare hvis for $\forall \mathbf{x} \in M^* : D_k(\mathbf{x}) = \mathbf{0}$.□

Teorem :

Gitt en modul M over en ring R. La D_r være en derivasjonsoperatoroperator for R, og la D_m være en derivasjonsoperator for M. Da eksisterer det en derivasjonsoperator D_k for komodulen $M^* = L(M; R)$. Vi vet videre at siden D_m har følgende egenskap:

$$D_m(\mathbf{b_i}) = e_i^j * \mathbf{b_j}$$

der $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ er basisen til M, vil den deriverte av den duale basis $B' = \{\mathbf{b^1}, \ldots, \mathbf{b^d}\}$ til M^* kunne uttrykkes som:

$$D_k(\mathbf{b^i}) = -e^i_j * \mathbf{b^j}$$

□

Bevis :

Beviset følger direkte fra egenskapene til D_k:

$$D_r(\mathbf{z}(\mathbf{x})) = (D_k(\mathbf{z}))(\mathbf{x}) + \mathbf{z}(D_m(\mathbf{x}))$$

og at den deriverte av den duale basis kan skrives som en lineaer kombinasjon:

$$D_k(\mathbf{b^i}) = f^i_j * \mathbf{b^j}$$

Vi kan utlede følgende:

$$\begin{array}{lcl}
D_r(\mathbf{b^i}(\mathbf{b_j})) & = & D_k(\mathbf{b^i})(\mathbf{b_j}) + \mathbf{b^i}(D_m(\mathbf{b_j})) \\
D_r(\delta^i_j) & = & f^i_p * \mathbf{b^p}(\mathbf{b_j}) + \mathbf{b^i}(e^q_j * \mathbf{b_q}) \\
0 & = & f^i_p * \delta^p_j + e^q_j * \delta^i_q \\
0 & = & f^i_j + e^i_j \\
f^i_j & = & -e^i_j
\end{array}$$

Konklusjonen blir derfor:

$$D_k(\mathbf{b^i}) = -e^i_j * \mathbf{b^j}$$

Teorem :

Gitt en modul M over en ring R. La D_r være en derivasjonsoperatoroperator for R, og la D_m være en derivasjonsoperatoroperator for M. Da er derivasjonsoperatoren D_k for komodulen M^* gitt som:

$$D_k(\mathbf{x}) = [D_r(x_j) - x_i * e^i_j] * \mathbf{b^j}$$

der e^i_j er slik at $D_m(\mathbf{b_i}) = e^j_i * \mathbf{b_j}$.□

Bevis :

Beviset følger direkte fra definisjonen for en derivasjonsoperator D_k for en komodul:

$$\begin{aligned}
D_k(\mathbf{x}) &= D_k(x_i * \mathbf{b^i}) \\
&= D_r(x_i) * \mathbf{b^i} + x_i * D_k(\mathbf{b^i}) \\
&= D_r(x_i) * \mathbf{b^i} + x_i * (-e^i_j * \mathbf{b^j}) \\
&= D_r(x_i) * \delta^i_j * \mathbf{b^j} - x_i * e^i_j * \mathbf{b^j} \\
&= (D_r(x_j) - x_i * e^i_j) * \mathbf{b^j}
\end{aligned}$$

Dette teoremet er svært viktig for oss, for det sier at hvis vi har en derivasjonsoperator for en ring R og en modul M over R, kan vi alltid finne den tilhørende derivasjonsoperatoren for komodulen M^*. Teoremet gir oss dessuten en algoritme for hvordan vi skal konstruere derivasjonsoperatoren D_k til M^*.

5.4 Tensorderivasjon

Gitt en modul M over en ring R. La D_r være en derivasjonsoperator for R, og la D_m være en derivasjonsoperator for M. En derivasjonsoperator D_t på $T(M)$ er en funksjon som for alle $r, s \geq 0$ er definert som et lineært kart $D_t : T^r_s(M) \rightarrow T^r_s(M)$, slik at for $\forall t \in T^r_s(M)$:

$$D_r(\mathbf{t}((\mathbf{a_1}, \ldots, \mathbf{a_r}, \mathbf{x_1}, \ldots, \mathbf{x_s}))) =$$

$$D_t(\mathbf{t})((\mathbf{a_1}, \ldots, \mathbf{a_r}, \mathbf{x_1}, \ldots, \mathbf{x_s})) +$$

$$\sum_{j=1}^{r} t((\mathbf{a_1}, \ldots, D_k(\mathbf{a_j}), \ldots, \mathbf{a_r}, \mathbf{x_1}, \ldots, \mathbf{x_s})) +$$

$$\sum_{j=1}^{s} t((\mathbf{a_1}, \ldots, \mathbf{a_r}, \mathbf{x_1}, \ldots, D_m(\mathbf{x_j}), \ldots * \mathbf{x_s}))$$

for alle $a_i \in M^*$ og alle $x_i \in M$.□

Vi ser ut fra definisjonen at D_t er entydig definert hvis vi har gitt D_r og D_m (siden D_k og er entydig definert ut fra D_r og D_m). Vi sier at D_t er triviell hvis for $\forall \mathbf{t} \in T^r_s(M)$: $D_t(\mathbf{t}) = \mathbf{0}(\mathbf{t})$. Vi kan videre merke oss fra definisjonen over at for $T^0_0(M)$ er D_t definert som ekvivalent med D_r. Vi kan trekke liknende konklusjoner for $T^1_0(M)$ og $T^0_1(M)$.

Teorem :

Gitt to tensorer, $\mathbf{t_1} \in T^{r1}_{s1}(M)$ og $\mathbf{t_2} \in T^{r2}_{s2}(M)$, vet vi at D_t distribuerer som følgende over tensorprodukt:

$$D_t(\mathbf{t_1} * \mathbf{t_2}) = D_t(\mathbf{t_1}) * \mathbf{t_2} + \mathbf{t_1} * D_t(\mathbf{t_2})$$

□

Bevis :

Beviset følger direkte fra definisjonen av tensorderivasjon, tensorprodukt og indreprodukt, og vi ser derfor kun grunntrekket ved beviset. La $\mathbf{t_1} \in T^{r_1}_{s_1}(M)$, $\mathbf{t_2} \in T^{r_2}_{s_2}(M)$, $\mathbf{t_3} \in T^{s_1}_{r_1}(M)$ og $\mathbf{t_4} \in T^{s_2}_{r_2}(M)$. La videre $\mathbf{t_3}$ være lik $\mathbf{a_1} * \ldots * \mathbf{a_{r_1}} * \mathbf{v_1} * \ldots * \mathbf{v_{s_1}}$ og $\mathbf{t_4}$ være lik $\mathbf{b_1} * \ldots * \mathbf{a_{r_2}} * \mathbf{w_1} * \ldots * \mathbf{v_{s_2}}$, der $\mathbf{a_i}, \mathbf{b_i} \in T^0_1(M)$ og $\mathbf{v_i}, \mathbf{w_i} \in T^1_0(M)$. Da har vi at:

$$
\begin{array}{rcl}
D_r((\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{t_4})) & = & D_t(\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{t_4}) + \\
& & \sum_{j=1}^{r1} (\mathbf{t_1} * \mathbf{t_2})(\mathbf{a_1} * \ldots * D_k(\mathbf{a_j}) * \ldots * \mathbf{a_{r_1}} * \mathbf{v_1} * \ldots * \mathbf{v_{s_1}} * \mathbf{t_4}) + \\
& & \sum_{j=1}^{s1} (\mathbf{t_1} * \mathbf{t_2})(\mathbf{a_1} * \ldots * \mathbf{a_{r_1}} * \mathbf{v_1} * \ldots * D_m(\mathbf{v_j}) * \ldots * \mathbf{v_{s_1}} * \mathbf{t_4}) + \\
& & \sum_{j=1}^{r2} (\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{b_1} * \ldots * D_k(\mathbf{b_j}) * \ldots * \mathbf{b_{r_1}} * \mathbf{w_1} * \ldots * \mathbf{w_{s_2}}) + \\
& & \sum_{j=1}^{s2} (\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{b_1} * \ldots * \mathbf{b_{r_1}} * \mathbf{w_1} * \ldots * D_m(\mathbf{w_j}) * \ldots * \mathbf{w_{s_2}}) \\
& \Updownarrow & \\
D_r(\mathbf{t_1}(\mathbf{t_3}) * \mathbf{t_2}(\mathbf{t_4})) & = & D_t(\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{t_4}) + \\
& & \sum_{j=1}^{r1} \mathbf{t_1}(\mathbf{a_1} * \ldots * D_k(\mathbf{a_j}) * \ldots * \mathbf{a_{r_1}} * \mathbf{v_1} * \ldots * \mathbf{v_{s_1}}) * \mathbf{t_2}(\mathbf{t_4}) + \\
& & \sum_{j=1}^{s1} \mathbf{t_1}(\mathbf{a_1} * \ldots * \mathbf{a_{r_1}} * \mathbf{v_1} * \ldots * D_m(\mathbf{v_j}) * \ldots * \mathbf{v_{s_1}}) * \mathbf{t_2}(\mathbf{t_4}) + \\
& & \mathbf{t_1}(\mathbf{t_3}) * \sum_{j=1}^{r2} \mathbf{t_2}(\mathbf{b_1} * \ldots * D_k(\mathbf{b_j}) * \ldots * \mathbf{b_{r_1}} * \mathbf{w_1} * \ldots * \mathbf{w_{s_2}}) + \\
& & \mathbf{t_1}(\mathbf{t_3}) * \sum_{j=1}^{s2} \mathbf{t_2}(\mathbf{b_1} * \ldots * \mathbf{b_{r_1}} * \mathbf{w_1} * \ldots * D_m(\mathbf{w_j}) * \ldots * \mathbf{w_{s_2}}) \\
& \Updownarrow & \\
D_t(\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{t_4}) & = & D_t(\mathbf{t_1})(\mathbf{t_3}) * \mathbf{t_2}(\mathbf{t_4}) + \mathbf{t_1}(\mathbf{t_3}) * D_t(\mathbf{t_2})(\mathbf{t_4}) \\
& \Updownarrow & \\
D_t(\mathbf{t_1} * \mathbf{t_2})(\mathbf{t_3} * \mathbf{t_4}) & = & (D_t(\mathbf{t_1}) * \mathbf{t_2})(\mathbf{t_3} * \mathbf{t_4}) + (\mathbf{t_1} * D_t(\mathbf{t_2}))(\mathbf{t_3} * \mathbf{t_4})
\end{array}
$$

som til slutt vil føre til at:

$$D_t(\mathbf{t_1} * \mathbf{t_2}) = D_t(\mathbf{t_1}) * \mathbf{t_2} + \mathbf{t_1} * D_t(\mathbf{t_2})$$

□

Dette teoremet er viktig for oss siden vi nå kan gi en ekvivalent definisjon av derivasjonsoperatorer for tensorer som er lettere å spesifisere algebraisk. Vi benytter oss i denne definisjonen av operasjonene som ble gitt i den algebraiske spesifikasjonen for tensorer. Det gjør at vi lettere får uttrykt den implisitte typekonverteringen som er i den matematiske definisjonen:

Alternativ definisjon:

Gitt en modul *Modul* over en ring *Ring*. La D_m være en derivasjonsoperator for sorten *Modul* og la D_r være en derivasjonsoperator for sorten *Ring*. En derivasjonsoperator D_t på sorten *Tensor* er en operasjon som har følgende egenskaper:

```
operations
   Dr ( _ ) : Ring     -> Ring
```

```
   Dm ( _ ) : Modul    -> Modul
   Dt ( _ ) : Tensor   -> Tensor
variables   a       : Ring
            m       : Modul
            t1, t2  : Tensor
equations
   Dt ( tensor ( a ) )          == tensor ( Dr ( a ) )
   Dt ( tensor ( m ) )          == tensor ( Dm ( m ) )
   Dt ( t1 * t2      )          == Dt ( t1 ) * t2 + t1 * Dt ( t2 )
   antKomoduler ( Dt ( t1 ) )   == antKomoduler ( t1 )
   antModuler   ( Dt ( t1 ) )   == antModuler   ( t1 )
end
```

□

Teorem :

Hvis vi har gitt en derivasjonsoperator D_r for en ring R og en derivasjonsoperator D_m for en modul M over R, kan vi entydig konstruere en derivasjonsoperator D_t for tensorrommet $T^r_s(M)$ som oppfyller kravene for en derivasjonsoperator for tensorer.

Bevis :

Beviset er ganske omfattende, men følger akkurat samme form som utledningen av derivasjonsoperatoren D_k for det duale rom til modulen M. Jeg vil derfor ikke gjøre beviset her, men bare vise resultatet. Vi benytter oss her av faktorene $\{e^i_j\}$ som er gitt ved:

$$D_m(\mathbf{b_i}) = e^j_i * \mathbf{b_j}$$

og basiselementene $\{\mathbf{b_i}\}$ for M, og basisen $\{\mathbf{b^j}\}$ for M^*. Vi kan da utlede at de deriverte av basiselementene $(\mathbf{b_{i_1}}, \ldots, \mathbf{b_{i_r}}, \mathbf{b^{j_1}}, \ldots, \mathbf{b^{j_s}}) \in T^r_s(M)$ kan finnes med formelen:

$$D_t((\mathbf{b_{i_1}}, \ldots, \mathbf{b_{i_r}}, \mathbf{b^{j_1}}, \ldots, \mathbf{b^{j_s}})) =$$

$$((e^{p_1}_{i_1} * \delta^{p_2,\ldots,p_r}_{i_2,\ldots,i_r} + \ldots + \delta^{p_1,\ldots,p_{r-1}}_{i_1,\ldots,i_{r-1}} * e^{p_r}_{i_r}) * \delta^{j_1,\ldots,j_s}_{q_1,\ldots,q_s}$$
$$-(e^{j_1}_{q_1} * \delta^{j_2,\ldots,j_s}_{q_2,\ldots,q_q} + \ldots + \delta^{j_1,\ldots,j_{s-1}}_{q_1,\ldots,q_{s-1}} * e^{j_s}_{q_s}) * \delta^{p_1,\ldots,p_r}_{i_1,\ldots,i_r})$$
$$*(\mathbf{b_{p_1}}, \ldots, \mathbf{b_{p_r}}, \mathbf{b^{q_1}}, \ldots, \mathbf{b^{q_s}})$$

der :

$$\delta^{j_1,\ldots,j_s}_{q_1,\ldots,q_s} = \delta^{j_1}_{q_1} * \ldots * \delta^{j_s}_{q_s}$$

Dette er en svært viktig sammenheng for oss, siden den sier at en derivasjonsoperator D_t for tensorer er entydig bestemt av derivasjonsoperatorene D_r og D_m for ringen og modulen som tensorene er konstruert over. Formelen over kan dessuten brukes for å finne derivasjonsoperatoren D_t. Vi kan utlede følgende algoritme for å finne den deriverte av en tensor $\mathbf{t} \in T_s^r(E)$:

$$
\begin{aligned}
D_t(\mathbf{t}) &= D_t(t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} * (\mathbf{b_{i_1}},\ldots,\mathbf{b_{i_r}},\mathbf{b^{j_1}},\ldots,\mathbf{b^{j_s}})) \\
&= D_r(t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s}) * (\mathbf{b_{i_1}},\ldots,\mathbf{b_{i_r}},\mathbf{b^{j_1}},\ldots,\mathbf{b^{j_s}}) + t^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} * D_t((\mathbf{b_{i_1}},\ldots,\mathbf{b_{i_r}},\mathbf{b^{j_1}},\ldots,\mathbf{b^{j_s}})) \\
&\Updownarrow \\
D_t(\mathbf{t}) &= (D_r(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * e^{p_1}_{i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * e^{p_r}_{i_r} - \\
&\quad t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * e^{j_1}_{q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * e^{j_s}_{q_s}) * (\mathbf{b_{p_1}},\ldots,\mathbf{b_{p_r}},\mathbf{b^{q_1}},\ldots,\mathbf{b^{q_s}})
\end{aligned}
$$

Vi ser raskt at konstruksjonen av D_k kun er et spesialtilfelle av formelen over. Formelen over gir oss dessuten direkte koordinatrepresentasjonen til den deriverte av en tensor. Denne formelen kommer vi til å benytte mye senere når vi skal implementere noen viktige derivasjonsoperatorer.

5.5 Algebraisk spesifikasjon

Ved hjelp av definisjonene over er det nå lett å utvide vår algebraiske spesifikasjon av ring, modul, komodul og tensor til og å inneholde en derivasjonsoperator. Jeg skriver ikke her opp en komplett spesifikasjon, men viser grunntrekkene, sammen med de nye operasjonene og likningene for hver spesifikasjon:

```
//////////////////////////////////////////////////////
//                                                  //
//                RING-DERIVASJON                   //
//                                                  //
//////////////////////////////////////////////////////

specification RING-DERIVASJON
parameters
   sort        Ring
   operations
      0        :               -> Ring
      1        :               -> Ring
      - _      : Ring          -> Ring
      _ + _    : Ring * Ring   -> Ring
      _ * _    : Ring * Ring   -> Ring
      Dr ( _ ) : Ring          -> Ring
specifies
   variables a, b : Ring
   equations
      RING ( Ring, 0, 1, -, +, * )
```

```
      Dr ( a + b ) == Dr ( a ) + Dr ( b )
      Dr ( a * b ) == Dr ( a ) * b + a * Dr ( b )
end specification


///////////////////////////////////////////////////
//                                               //
//              MODUL-DERIVASJON                 //
//                                               //
///////////////////////////////////////////////////

specification MODUL-DERIVASJON
include BOOL,HELTALL
parameters
   sort          Ring, Modul
   operations
      0                 :                                             -> Ring
      1                 :                                             -> Ring
      - _               : Ring                                        -> Ring
      _ + _             : Ring * Ring                                 -> Ring
      _ * _             : Ring * Ring                                 -> Ring
      Dr ( _ )          : Ring                                        -> Ring

      0                 :                                             -> Modul
      _ + _             : Modul * Modul                               -> Modul
      - _               : Modul                                       -> Modul
      _ * _             : Ring * Modul                                -> Modul
      dimensjon ( _ )   : Modul                                       -> Heltall
      koordinat ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
      basis     ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1)) -> Modul
      Dm ( _ )          : Modul                                       -> Modul
specifies
   variables     v, w : Modul
                 a    : Ring
   equations
      RING-DERIVASJON ( Ring,
                        0, 1, -, +, *, Dr )
      MODUL           ( Ring, Modul,
                        0, 1, -, +, * ,
                        0, +, -, *,  dimensjon, koordinat, basis )
      Dm ( v + w ) == Dr ( v ) + Dr ( w )
      Dr ( a * v ) == Dr ( a ) * v + a * Dm ( v )
end specification



///////////////////////////////////////////////////
//                                               //
//             KOMODUL-DERIVASJON                //
//                                               //
///////////////////////////////////////////////////
```

```
specification KOMODUL-DERIVASJON
include BOOL,HELTALL
parameters
   sort          Ring, Modul, koModul
   operations
      0                 :                                            -> Ring
      1                 :                                            -> Ring
      - _               : Ring                                       -> Ring
      _ + _             : Ring * Ring                                -> Ring
      _ * _             : Ring * Ring                                -> Ring
      Dr ( _ )          : Ring                                       -> Ring

      0                 :                                            -> Modul
      _ + _             : Modul * Modul                              -> Modul
      - _               : Modul                                      -> Modul
      _ * _             : Ring * Modul                               -> Modul
      dimensjon ( _ )   : Modul                                      -> Heltall
      koordinat ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1))    -> Ring
      basis     ( _,_ ) : Modul * Heltall | (0<#2<=dimensjon(#1))    -> Modul
      Dm ( _ )          : Modul                                      -> Modul

      0                 :                                            -> koModul
      _ + _             : koModul * koModul                          -> koModul
      - _               : koModul                                    -> koModul
      _ * _             : Ring * koModul                             -> koModul
      _ ( _ )           : koModul * Modul                            -> Ring
      dimensjon ( _ )   : koModul                                    -> Heltall
      koordinat ( _,_ ) : koModul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
      basis     ( _,_ ) : koModul * Heltall | (0<#2<=dimensjon(#1)) -> koModul
      _ ( _ )           : koModul * Modul                            -> Ring
      Dk ( _ )          : koModul                                    -> koModul

specifies
   variables  v : koModul
              w : Modul
   equations
      MODUL-DERIVASJON ( Ring, Modul, 0, 1, -, +, * , Dr,
                         0, +, -, *,
                         dimensjon, koordinat, basis, Dm  )
      KOMODUL           ( Ring, modul, koModul,
                         0, 1, -, +, * ,
                         0, +, -, *, dimensjon, koordinat, basis,
                         0, +, -, *, dimensjon, koordinat, basis, _ ( _ ) )

      Dr ( v ( w ) ) == Dk ( v )( w ) + v ( Dm ( w ) )
end specification


////////////////////////////////////////////////////
//                                              //
```

```
//              TENSOR-DERIVASJON                    //
//                                                   //
///////////////////////////////////////////////////////

specification TENSOR-DERIVASJON
include BOOL, HELTALL
parameters
   sort         Ring, Modul, koModul, Tensor
   operations
      0                  :                                           -> Ring
      1                  :                                           -> Ring
      - _                : Ring                                      -> Ring
      _ + _              : Ring * Ring                               -> Ring
      _ * _              : Ring * Ring                               -> Ring
      Dr ( _ )           : Ring                                      -> Ring


      0                  :                                           -> Modul
      _ + _              : Modul * Modul                             -> Modul
      - _                : Modul                                     -> Modul
      _ * _              : Ring * Modul                              -> Modul
      dimensjon ( _ )    : Modul                                     -> Heltall
      koordinat ( _,_ )  : Modul * Heltall | (0<#2<=dimensjon(#1))   -> Ring
      basis     ( _,_ )  : Modul * Heltall | (0<#2<=dimensjon(#1))   -> Modul
      Dm        ( _ )    : Modul                                     -> Modul

      0                  :                                           -> koModul
      _ + _              : koModul * koModul                         -> koModul
      - _                : koModul                                   -> koModul
      _ * _              : Ring * koModul                            -> koModul
      _ ( _ )            : koModul * Modul                           -> Ring
      dimensjon ( _ )    : koModul                                   -> Heltall
      koordinat ( _,_ )  : koModul * Heltall | (0<#2<=dimensjon(#1)) -> Ring
      basis     ( _,_ )  : koModul * Heltall | (0<#2<=dimensjon(#1)) -> koModul
      _ ( _ )            : koModul * Modul                           -> Ring
      Dk ( _ )           : koModul                                   -> koModul

      ring          ( _ ) : Tensor | erRing    (#1)                  -> Ring
      modul         ( _ ) : Tensor | erModul   (#1)                  -> Modul
      komodul       ( _ ) : Tensor | erKomodul(#1)                   -> koModul
      antKomoduler ( _ ) : Tensor                                    -> Heltall
      antModuler    ( _ ) : Tensor                                   -> Heltall

      tensor ( _ )        : Ring                                     -> Tensor
      tensor ( _ )        : Modul                                    -> Tensor
      tensor ( _ )        : koModul                                  -> Tensor

      0 ( _ )             : Tensor                                   -> Tensor
      _ + _               : Tensor * Tensor                          -> Tensor
      - _                 : Tensor                                   -> Tensor
      _ * _               : Ring * Tensor                            -> Tensor
```

```
    dimensjon ( _ )    : Tensor                                   -> Heltall
    koordinat ( _,_ )  : Tensor * Heltall | (0<#2<=dimensjon(#1)) -> Ring
    basis     ( _,_ )  : Tensor * Heltall | (0<#2<=dimensjon(#1)) -> Tensor

    _ * _              : Tensor * Tensor                          -> Tensor
    < _,_ >            : Tensor * Tensor                          -> Tensor
    kontraksjon ( _ )  : Tensor | kanKontraktere ( #1 )           -> Tensor
    Kronecker          :                                          -> Tensor
    Dt ( _ )           : Tensor                                   -> Tensor

    erRing          ( _ )    : Tensor                             -> Bool
    erModul         ( _ )    : Tensor                             -> Bool
    erKomodul       ( _ )    : Tensor                             -> Bool
    sammeType       ( _,_ )  : Tensor * Tensor                    -> Bool
    gyldigIndreprod ( _, _ ) : Tensor * Tensor                    -> Bool
    kanKontraktere  ( _ )    : Tensor                             -> Bool
specifies
   variables  a      : Ring
              m      : Modul
              t1, t2 : Tensor
   equations
      KOMODUL-DERIVASJON  ( Ring, Modul, koModul,
                            0, 1, -, +, * , Dr,
                            0, +, -, *, dimensjon, koordinat, basis, Dm,
                            0, +, -, *, dimensjon, koordinat, basis, _ ( _ ), Dk )

      TENSOR              ( Ring, Modul, koModul, Tensor,
                            0, 1, -, +, * ,
                            0, +, -, *, dimensjon, koordinat, basis,
                            0, +, -, *, dimensjon, koordinat, basis, _ ( _ ),
                            ring, modul, komodul, antKomoduler, antModuler,
                            tensor, tensor, tensor, 0, +, -, *, dimensjon,
                            koordinat, basis, *, <,>, kontraksjon, Kronecker,
                            erRing, erModul, erKomodul,
                            sammeType, gyldigIndreprod, kanKontraktere )

      Dt ( tensor ( a ) )         == tensor ( Dr ( a ) )
      Dt ( tensor ( m ) )         == tensor ( Dm ( m ) )
      Dt ( t1 * t2      )         == Dt ( t1 ) * t2 + t1 * Dt ( t2 )
      antKomoduler ( Dt ( t1 ) )  == antKomoduler ( t1 )
      antModuler   ( Dt ( t1 ) )  == antModuler   ( t1 )
end specification
```

Chapter 6

Deriverbare strukturer

Til nå har vi bare spesifisert hvilke egenskaper derivasjonsoperatorer skal ha, vi har ikke spesifisert noen operasjoner som inneholder disse egenskapene. I dette kapittelet skal vi se nærmere på noen strukturer som kan deriveres, og vi skal og spesifisere noen vanlige derivasjonsoperatorer for disse strukturene. Det er særlig to derivasjonsoperatorer som er mye brukt innenfor anvendt tensor algebra, det er *Lie derivasjon* og *kovariant derivasjon*, og disse derivasjonsoperatorene må derfor defineres.

De strukturene som vanligvis blir brukt når en definerer derivasjonsoperatorer for tensorer er *Banachrom* og *deriverbare manifolder*. Et typisk eksempel på dette er [1] som definerer tensorer over *tangentbunken* til en manifold. Vi kan ikke benytte denne innfallsvinkelen, siden en deriverbar manifold M og tangentbunken $T(M)$ over M er definert ut fra eksistensen av et *banachrom*. Et banachrom er et *vektorrom* over en *kropp* med noen ekstra egenskaper. Definisjonen av derivasjonsoperatorer på M baserer seg sterkt på bruk av divisjonsoperatoren til kroppen, og denne operatoren har vi ikke tilgjengelig i en Ring. Problemet for oss er at vi ikke kan snakke om lokale basiser og kartområder for en manifold. Den eneste strukturen som vi kan benytte er en modul, og det har da ikke noen mening å snakke om lokale basiser. I resten av denne oppgaven skal vi derfor begrense oss til å kun arbeide med moduler som har en fast basis (ikke lokale basiser). Denne begrensningen er ekvivalent med at vi begrenser oss til det som i [1] blir referert som *parallelliserbare mangfoldigheter*.

6.1 Deriverbare ringer

To derivasjonsoperatorer D_1 og D_2 for en ring R er *lineært uavhengige* dersom:

$$\forall a, b \in R : a * D_1(c) + b * D_2(c) = 0 \Rightarrow a = b = 0$$

der $c \in R$ er slik at enten $D_1(c)$ eller $D_2(c)$ er ulik ringkonstanten 0.□

Def. deriverbar ring

En ring R sammen med mengden $D_R = \{\frac{\partial}{\partial x^1}, \ldots, \frac{\partial}{\partial x^d}\}$ av alle lineært uavhengige ikke trivielle derivasjonsoperatorer for R kaller vi en *deriverbar ring*. Vi sier at R har d *dimensjoner*, og anvendelsen av derivasjonsoperator $\frac{\partial}{\partial x^i} \in D$ på et element $e \in R$ kalles *den partielle deriverte* av e i *dimensjon* i.□

Hvis en deriverbar ring R har dimensjon 0, betyr det at ringen kun har en derivasjonsoperator, og den er triviell. Dette er siden alle ringer har en triviell derivasjonsoperator.

I et programmeringsspråk kan vi betrakte en deriverbar ring som en abstrakt datatype med innebygde operasjoner for å derivere seg selv. Dette betyr at vi kan innkapsle implementasjonsdetaljene til derivasjonsoperatorene i ADT'en. Vi trenger derfor ikke tenke på hvordan vi skal beregne de partielle deriverte av en deriverbar ring, det eneste vi skal vente av en implementasjon er at operasjonene er tilgjengelige og oppfyller spesifikasjonene.

Eksempel :

Som et eksempel på en deriverbar ring, kan vi bruke mengden K av alle funksjoner fra d ($0 < d$) reelle tall R, til R. D.v.s.:

$$K = \{f : R^d \to R\}$$

Vi gir nå K ringegenskaper på følgende måte:

$$\begin{array}{lcl} \forall x_1, \ldots, x_d \in R & : & 0(x_1, \ldots, x_d) = 0 \\ \forall x_1, \ldots, x_d \in R & : & 1(x_1, \ldots, x_d) = 1 \\ \forall x_1, \ldots, x_d \in R, \forall a \in K & : & (-a)(x_1, \ldots, x_d) = -a(x_1, \ldots, x_d) \\ \forall x_1, \ldots, x_d \in R, \forall a, b \in K & : & (a+b)(x_1, \ldots, x_d) = a(x_1, \ldots, x_d) + b(x_1, \ldots, x_d) \\ \forall x_1, \ldots, x_d \in R, \forall a, b \in K & : & (a*b)(x_1, \ldots, x_d) = a(x_1, \ldots, x_d) * b(x_1, \ldots, x_d) \end{array}$$

I tillegg har vi den vanlige derivasjonsoperatoren $\frac{\partial}{\partial x^i} : K \to K$, som er den partielle deriverte for en funksjon $f : R^d \to R$ m.h.p. argument nr. i $(0 < i \leq d)$. Vi vet fra lineær algebra at $\frac{\partial}{\partial x^i}$ har følgende egenskaper :

- $\frac{\partial}{\partial x^i}(f+g) = \frac{\partial}{\partial x^i}(f) + \frac{\partial}{\partial x^i}(g)$
- $\frac{\partial}{\partial x^i}(f*g) = \frac{\partial}{\partial x^i}(f) * g + f * \frac{\partial}{\partial x^i}(g)$

Det betyr at $\frac{\partial}{\partial x^i}$ er tilfredstiller våre krav for en derivasjonsoperator for ringen K (vi forutsetter her at alle elementer $f \in K$ er uendelig mange ganger deriverbare).

6.2 Deriverbare moduler

Def. deriverbar modul

La R være en deriverbar ring, og la D_R være mengden av alle lineært uavhengige, ikke trivielle derivasjonsoperatorer for R. Vi definerer nå modulen $D[R]$ til å være mengden av alle derivasjonsoperatorer for R, der basisen til $D[R]$ er mengden D_R. Anvendelsen av et element $\mathbf{d} \in D[R]$ på et element $e \in R$ kaller vi den *retningsderiverte* til e i *retning* $\mathbf{d}$. Modulen $D[R]$ kaller vi den *deriverbare modulen* til R.□

Vi vet at alle derivasjonsoperatorer for en ring R kan skrives som en lineær kombinasjon av elementene i mengden D_R, og $D[R]$ må derfor inneholde alle mulige derivasjonsoperatorer for R. Modulegenskapene til $D[R]$ kan konstrueres ved å spesifisere operasjonene '0', '+', '-' og '*' på følgende måte:

$$\begin{array}{lcll}
\forall r \in R & : & \mathbf{0}(r) & == \; 0 \\
\forall \mathbf{D} \in D[R], \forall r \in R & : & (-\mathbf{D})(r) & == \; -\mathbf{D}(r) \\
\forall \mathbf{D1}, \mathbf{D2} \in D[R], \forall r \in R & : & (\mathbf{D1} + \mathbf{D2})(r) & == \; \mathbf{D1}(r) + \mathbf{D2}(r) \\
\forall \mathbf{D} \in D[R], \forall k, r \in R & : & (k * \mathbf{D})(r) & == \; k * \mathbf{D}(r)
\end{array}$$

Merk at elementet $\mathbf{0} \in D[R]$ er den trivielle derivasjonsoperatoren for R. Siden D_R er en basis for $D[R]$, kan vi og bevise at alle elementer i mengden $D[R]$ må være derivasjonsoperatorer for R. For å bevise at $D[R]$ kun inneholder <u>gyldige</u> derivasjonsoperatorer for R må vi vise at moduloperasjonene vi spesifiserte over ikke genererer elementer i $D[R]$ som ikke er derivasjonsoperatorer. Det betyr at vi måvis at for $\forall x, y, z \in R$, $\forall \frac{\partial}{\partial \mathbf{x^i}}, \frac{\partial}{\partial \mathbf{x^j}} \in D[R]$:

$$\begin{array}{lcll}
1 & : & (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x+y) & = \; (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x) + (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(y) \\
2 & : & (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x*y) & = \; (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x) * y + x * (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(y) \\
3 & : & (z * \frac{\partial}{\partial \mathbf{x^i}})(x+y) & = \; (z * \frac{\partial}{\partial \mathbf{x^i}})(x) + (z * \frac{\partial}{\partial \mathbf{x^i}})(y) \\
4 & : & (z * \frac{\partial}{\partial \mathbf{x^i}})(x*y) & = \; (z * \frac{\partial}{\partial \mathbf{x^i}})(x) * y + x * (z * \frac{\partial}{\partial \mathbf{x^i}})(y) \\
5 & : & (-\frac{\partial}{\partial \mathbf{x^i}})(x+y) & = \; (-\frac{\partial}{\partial \mathbf{x^i}})(x) + (-\frac{\partial}{\partial \mathbf{x^i}})(y) \\
6 & : & (-\frac{\partial}{\partial \mathbf{x^i}})(x*y) & = \; (-\frac{\partial}{\partial \mathbf{x^i}})(x) * y + x * (-\frac{\partial}{\partial \mathbf{x^i}})(y)
\end{array}$$

Beviset er omfattende men lett, og det holder derfor å skissere fremgangsmåten m.h.p. operasjonen '+':

$$\begin{array}{rcl}
(\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x+y) & = & \frac{\partial}{\partial \mathbf{x^i}}(x+y) + \frac{\partial}{\partial \mathbf{x^j}}(x+y) \\
& = & \frac{\partial}{\partial \mathbf{x^i}}(x) + \frac{\partial}{\partial \mathbf{x^i}}(y) + \frac{\partial}{\partial \mathbf{x^j}}(x) + \frac{\partial}{\partial \mathbf{x^j}}(y) \\
& = & \frac{\partial}{\partial \mathbf{x^i}}(x) + \frac{\partial}{\partial \mathbf{x^j}}(x) + \frac{\partial}{\partial \mathbf{x^i}}(y) + \frac{\partial}{\partial \mathbf{x^j}}(x) \\
& = & (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x) + (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(y)
\end{array}$$

og:

$$\begin{aligned}
(\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x * y) &= \frac{\partial}{\partial \mathbf{x^i}}(x * y) + \frac{\partial}{\partial \mathbf{x^j}}(x * y) \\
&= \frac{\partial}{\partial \mathbf{x^i}}(x) * y + x * \frac{\partial}{\partial \mathbf{x^i}}(y) + \frac{\partial}{\partial \mathbf{x^j}}(x) * y + x * \frac{\partial}{\partial \mathbf{x^j}}(y) \\
&= [\frac{\partial}{\partial \mathbf{x^i}}(x) + \frac{\partial}{\partial \mathbf{x^j}}(x)] * y + x * (\frac{\partial}{\partial \mathbf{x^i}}(y) + \frac{\partial}{\partial \mathbf{x^j}}(y)) \\
&= (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(x) * y + x * (\frac{\partial}{\partial \mathbf{x^i}} + \frac{\partial}{\partial \mathbf{x^j}})(y)
\end{aligned}$$

Dette betyr at alle derivasjonsoperatorer $\mathbf{D} \in D[R]$ kan skrives som en lineær kombinasjon av basiselementene $\{\frac{\partial}{\partial \mathbf{x^1}}, \ldots, \frac{\partial}{\partial \mathbf{x^d}}\}$, slik:

$$\mathbf{D} = D^i * \frac{\partial}{\partial \mathbf{x^i}} \quad , \quad 0 < i \leq d, D^i \in R$$

6.3 Deriverbare komoduler

Gitt en deriverbar ring R og den deriverbare modulen $D[R]$ over R. Siden $D[R]$ er en modul over R, vet vi at det finnes en dual modul $D[R]^*$ over R. Vi definerer den duale basisen til $D[R]^*$ å være $\{\mathbf{dx^1}, \ldots, \mathbf{dx^d}\}$, og gir basisen følgende egenskap:

$$\mathbf{dx^i}(\frac{\partial}{\partial \mathbf{x^j}}) = \delta^i_j$$

Vi sier da at $D[R]^*$ er den *deriverbare komodulen* til $R\Box$.

6.4 Deriverbare tensorer

Gitt en deriverbar ring R, sammen med den deriverbare modulen $D[R]$ over R og den deriverbare komodulen $D[R]^*$ over R. Vi definerer nå mengden:

$$T^r_s(R) = T^r_s(D[R]) = L((D[R]^*)^r * D[R]; R)$$

for alle $r, s \geq 0$. Elementene i mengden $T^r_s(R)$ kaller vi *deriverbare tensorer* over den deriverbare ringen $R.\Box$

Merk her at alle deriverbare tensorer og er tensorer etter vår tidligere definisjon. Alle tensoroperasjoner vi har beskrevet tidligere gjelder derfor og for deriverbare tensorer. Vi skal nå videre beskrive noen derivasjonsoperatorer for deriverbare moduler og deriverbare tensorer.

6.5 Lie derivasjon

6.5.1 Lie derivasjon for deriverbar ring

Gitt en deriverbar ring R, sammen med den deriverbare modulen $D[R]$ over R. Vi definerer den *Lie deriverte* (eller retningsderiverte) av $f \in K$ i retning $\mathbf{w} \in D[K]$ til å være :

$$\mathcal{L}_{\mathbf{w}}(f) = w^i * \frac{\partial}{\partial \mathbf{x^i}}(f)$$

□

Det kan vises at denne definisjonen er uavhengig av en fast basis for modulen $D[R]$. Man kan derfor gi en ekvivalent definisjon av den Lie deriverte som er uavhengig av en fast basis:

- $\mathcal{L}_{\frac{\partial}{\partial \mathbf{x^i}}}(f) == \frac{\partial}{\partial x^i}(f)$
- $\mathcal{L}_{\mathbf{v}+\mathbf{w}}(f) == \mathcal{L}_{\mathbf{v}}(f) + \mathcal{L}_{\mathbf{w}}(f)$
- $\mathcal{L}_{k*\mathbf{w}}(f) == k * \mathcal{L}_{\mathbf{w}}(f)$

for $\forall \mathbf{v}, \mathbf{w} \in D[R]$, og for $\forall k, f \in R$.

6.5.2 Lie derivasjon for deriverbar modul

Gitt en deriverbar ring R, sammen med den deriverbare modulen $D[R]$ over R. Den *Lie deriverte* av et element $\mathbf{w} \in D[R]$ i retning $\mathbf{v} \in D[R]$ er entydig definert til å være :

$$\mathcal{L}_{\mathbf{v}}(\mathbf{w}) = [\mathbf{v}, \mathbf{w}]$$

der $[\mathbf{v}, \mathbf{w}]$ er entydig definert som det elementet i $D[R]$ som tilfredstiller følgende krav :

$$\mathcal{L}_{[\mathbf{v},\mathbf{w}]}(f) = \mathcal{L}_{\mathbf{v}}(\mathcal{L}_{\mathbf{w}}(f)) - \mathcal{L}_{\mathbf{w}}(\mathcal{L}_{\mathbf{v}}(f))$$

for $\forall f \in R$. $[\mathbf{v}, \mathbf{w}]$ kalles ofte *Lie-Jakobi* braketten.□

Det kan vises (se f.eks. [1]) at $\mathcal{L}_{\mathbf{w}}(\mathbf{v})$ kan beregnes på følgende måte:

$$\begin{aligned}
\mathcal{L}_{\mathbf{w}}(\mathbf{v}) &= (\mathcal{L}_{\mathbf{w}}(v^i) - \mathcal{L}_{\mathbf{v}}(w^i)) * \frac{\partial}{\partial \mathbf{x^i}} \\
&\Updownarrow \\
\mathcal{L}_{\mathbf{w}}(\mathbf{v}) &= (w^j * \frac{\partial}{\partial x^j}(v^i) - v^j * \frac{\partial}{\partial x^j}(w^i)) * \frac{\partial}{\partial \mathbf{x^i}}
\end{aligned}$$

og at $\mathcal{L}_{\mathbf{w}}$ er en gyldig derivasjonsoperator for $D[R]$.

6.5.3 Lie derivasjon for deriverbar komodul

Tidligere så vi at hvis vi har en derivasjonsoperator D_r for en ring R, og en derivasjonsoperator D_m for en modul M over R, kan vi konstruere den tilsvarende derivasjonsoperatoren for M^*. Dette benytter vi oss av når vi nå skal finne den Lie deriverte av elementer i $D[R]^*$. For å klare dette må vi finne konstantene $E = \{e^i_j\}$ som er gitt ved:

$$\mathcal{L}_{\mathbf{w}}(\frac{\partial}{\partial \mathbf{x^i}}) = e_i^j * \frac{\partial}{\partial \mathbf{x^j}}$$

for alle $\frac{\partial}{\partial \mathbf{x^i}}$ som er basiselementer i $D[R]$. Dette klarer vi lett ved formelen foran, og får at:

$$\begin{array}{lcl} \mathcal{L}_{\mathbf{w}}(\frac{\partial}{\partial \mathbf{x^i}}) & = & -\frac{\partial}{\partial x^i}(w^j) * \frac{\partial}{\partial \mathbf{x^j}} \\ & \Updownarrow & \\ e_j^i & = & -\frac{\partial}{\partial x^j}(w^i) \end{array}$$

Ved hjelp av likningen for konstruksjon av derivasjonsoperator D_k for en komodul, får vi derfor at den Lie deriverte for $D[R]^*$ er:

$$\begin{array}{lcl} D_k(\mathbf{w}) & = & (D_r(w_j) - w_i * e_j^i) * \mathbf{dx^j} \\ & \Updownarrow & \\ \mathcal{L}_{\mathbf{v}}(\mathbf{w}) & = & (\mathcal{L}_{\mathbf{v}}(w_j) - w_i * e_j^i) * \mathbf{dx^j} \\ & \Updownarrow & \\ \mathcal{L}_{\mathbf{v}}(\mathbf{w}) & = & (v^i * \frac{\partial}{\partial x^i}(w^j) + w_i * \frac{\partial}{\partial x^j}(v^i)) * \mathbf{dx^j} \end{array}$$

for $\forall \mathbf{v} \in D[R], \forall \mathbf{w} \in D[R]^*$, og $\{\mathbf{dx^j}\}$ er basiselementer for $D[R]^*$.

6.5.4 Lie derivasjon for deriverbare tensorer

Gitt en deriverbar ring R. Vi har tidligere vist at hvis vi har gitt en derivasjonsoperator D_r for R og en derivasjonsoperator D_m for en modul M over R, kan vi alltid finne en derivasjonsoperator D_t for tensorrommet $T_s^r(M)$. Vi har definert den Lie deriverte for en deriverbar ring R og for den deriverbare modulen $D[R]$ over R. Vi vet derfor at det finnes en Lie derivasjon for alle deriverbare tensorrom $T_s^r(R)$, og denne derivasjonsoperatoren er entydig gitt av R og $D[R]$. Vi kan derfor bruke resultatet foran for å finne den Lie deriverte til en tensor $\mathbf{t} \in T_s^r(R)$ i retning $\mathbf{w} \in D[R]$. Vi fant at tidligere at:

$$e_j^i = -\frac{\partial}{\partial x^j}(w^i)$$

for den lie deriverte, og har fra tidligere følgende formel:

$$\begin{array}{lcl} D_t(\mathbf{t}) & = & (D_r(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * e^{p_1}_{i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * e^{p_r}_{i_r} - \\ & & t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * e^{j_1}_{q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * e^{j_s}_{q_s}) * (\mathbf{b_{p_1}},\ldots,\mathbf{b_{p_r}},\mathbf{b^{q_1}},\ldots,\mathbf{b^{q_s}}) \end{array}$$

og finner derfor ved innsetting:

$$\mathcal{L}_{\mathbf{w}}(\mathbf{t}) = (w^i * \frac{\partial}{\partial x^i}(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) - t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * \frac{\partial}{\partial x^{i_1}}(w^{p_1}) - \ldots - t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * \frac{\partial}{\partial x^{i_r}}(w^{p_r})$$

$$+t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * \frac{\partial}{\partial x^{q_1}}(w^{j_1}) + \ldots + t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * \frac{\partial}{\partial x^{q_s}}(w^{j_s}))*$$

$$(\frac{\partial}{\partial \mathbf{x}^{\mathbf{p_1}}}, \ldots, \frac{\partial}{\partial \mathbf{x}^{\mathbf{p_r}}}, \mathbf{dx}^{\mathbf{q_1}}, \ldots, \mathbf{dx}^{\mathbf{q_s}})$$

for alle $\mathbf{t} \in T^r_s(R)$ og alle $\mathbf{w} \in D[R]$.

Dette har stor praktisk betydning for oss, siden vi nå vet hvordan vi skal beregne den Lie deriverte til en vilkårlig tensor hvis vi har tilgjengelig derivasjonsoperatorene som beregner de Lie deriverte for en deriverbar ring R og den deriverbare modulen $D[R]$.

6.6 Partiell derivasjon

6.6.1 Christoffel-symboler

Vi ønsker å utvide definisjonen av partiell derivasjon til og å gjelde for en deriverbar modul $D[R]$. Vi må derfor definere den partielle deriverte for alle modulelement $\mathbf{w} \in D[R]$. Det gjør vi ved først å spesifisere hvordan basiselementene $\{\frac{\partial}{\partial \mathbf{x}^{\mathbf{1}}}, \ldots, \frac{\partial}{\partial \mathbf{x}^{\mathbf{d}}}\}$ til $D[R]$ deriveres. Dette er siden:

$$\begin{aligned}
\tfrac{\partial}{\partial x^i}(\mathbf{w}) &= \tfrac{\partial}{\partial x^i}(w^j * \tfrac{\partial}{\partial \mathbf{x}^{\mathbf{j}}}) \\
&= \tfrac{\partial}{\partial x^i}(w^j) * \tfrac{\partial}{\partial \mathbf{x}^{\mathbf{j}}} + w^j * \tfrac{\partial}{\partial x^i}(\tfrac{\partial}{\partial \mathbf{x}^{\mathbf{j}}})
\end{aligned}$$

Siden den partielle deriverte av et modulelement og er et modulelement, må den partielle deriverte av et basiselement kunne skrives som en lineær kombinasjon:

$$\frac{\partial}{\partial x^i}(\frac{\partial}{\partial \mathbf{x}^{\mathbf{j}}}) = \Gamma^q_{i,j} * \frac{\partial}{\partial \mathbf{x}^{\mathbf{q}}}$$

der $\Gamma^q_{i,j} \in R$ for alle $0 < i, j, q \leq d$. Av diskusjonen tidligere vet vi da at den partielle deriverte er en gyldig derivasjonsoperator. I definisjonen av den Lie deriverte til et modulelement entydig gitt av kravet:

$$\mathcal{L}_{[\mathbf{v},\mathbf{w}]}(f) = \mathcal{L}_{\mathbf{v}}(\mathcal{L}_{\mathbf{w}}(f)) - \mathcal{L}_{\mathbf{w}}(\mathcal{L}_{\mathbf{v}}(f))$$

Ved partiell derivasjon har vi ikke noen krav som entydig definerer verdiene til $\Gamma^q_{i,j}$, de må gis eksplisitt. Tallene $\Gamma^q_{i,j}$ kalles ofte *euklidisk forbindelse*, eller *Christoffel-symboler*. Det kan vises at elementene $\Gamma^q_{i,j}$ ikke har tensoregenskaper, de er for oss bare 'en mengde elementer' som er gitt. Vi kan derfor ikke påstå noen ytterlige algebraiske egenskaper for $\Gamma^q_{i,j}$. De er bare gitt som en spesifikasjon av hvordan den partielle deriverte av basisen til $D[R]$ oppfører seg.

6.6.2 Den metriske tensor

Det er likevel en sammenheng mellom tensorer og Christoffel-symbolene, og det er gitt ved noe som kalles den *metriske tensor*.

Den metriske tensor er heller ikke en tensor som generellt kan beregnes for alle moduler $D[R]$, men må være gitt eksplisitt. Den metriske tensoren blir oftest referert ved symbolet $\mathbf{g}$, og er av type (0, 2). Denne tensoren er spesiell, da vi krever at det skal finnes en 'invers' metrisk tensor, $\mathbf{g}^*$ av type (2, 0) som har følgende egenskap:

- $\forall \mathbf{v} \in T_0^1(R) :< \mathbf{g}^*, < \mathbf{g}, \mathbf{v} >>= \mathbf{v}$
- $\forall \mathbf{w} \in T_1^0(R) :< \mathbf{g}, < (\mathbf{g}^*, \mathbf{w} >>= \mathbf{w}$

Det finnes vanligvis mange tensorer som oppfyller dette kravet, og $\mathbf{g}$ må derfor være gitt 'utenfor' vårt system. Fysisk sett blir $\mathbf{g}$ brukt for å gi oss et 'lengdemål' for elementer i $D[R]$, og blir vanligvis bare brukt i felter som kan representeres i såkalte kartesiske koordinatsystem.

Sammenhengen mellom g, g^* og $\Gamma^q_{i,j}$ er at hvis vi har gitt g og g^* kan vi alltid beregne $\Gamma^q_{i,j}$, men ikke omvendt. Vi skal ikke bevise dette her, men bare referere til følgende sammenheng tatt fra [4]:

$$\Gamma^n_{i,j} = \frac{1}{2} * g^{n,k} * (\frac{\partial}{\partial x^j}(g^*_{k,i}) + \frac{\partial}{\partial x^i}(g^*_{j,k}) - \frac{\partial}{\partial x^k}(g^*_{i,j}))$$

Det må presiseres at for å kunne si noe om den metriske tensor og $\Gamma^q_{i,j}$, må vi få tilført flere opplysninger enn vi har tilgjengelig kun ut fra $D[R]$. Det eksisterer felter der det ikke finnes noen metrisk tensor som har noen fornuftig fysisk mening. Det enenste vi derfor kan gjøre her er å påstå er at det finnes noe som kalles christoffel-symboler, som består av d^3 ringelementer, og indekseres ved hjelp av tre indekser.

6.6.3 Partiell derivasjon av deriverbare moduler, komoduler og tensorer

Gitt en deriverbar ring R, sammen med den deriverbare modulen $D[R]$ av derivasjons-operatorer for R. La $B = \{\frac{\partial}{\partial \mathbf{x^1}}, \ldots, \frac{\partial}{\partial \mathbf{x^d}}\}$ være basisen for $D[R]$. Vi har videre gitt christoffelsymbolene $\Gamma^q_{i,j}$ for $D[R]$. Da er den partielle deriverte av et element $\mathbf{w} \in D[R]$ definert på følgende måte :

$$\frac{\partial}{\partial x^i}(\mathbf{w}) = (\frac{\partial}{\partial x^i}(w^q) + w^j * \Gamma^q_{i,j}) * \frac{\partial}{\partial \mathbf{x^q}}$$

Siden vi, gitt en derivasjonsoperator for modulen $D[R]$, kan utvide derivasjonsoperatoren til og å gjelde komodulen $D[R]^*$ og $T_s^r(R)$, kan vi utlede at den partielle deriverte til et element $\mathbf{v} \in D[R]^*$ er:

$$\frac{\partial}{\partial x^i}(\mathbf{v}) = (\frac{\partial}{\partial x^i}(v_q) - v_j * \Gamma^j_{i,q}) * \mathbf{dx^q}$$

og for alle deriverbare tensorer $\mathbf{t} \in T_s^r(R)$ er den partielle deriverte:

$$\frac{\partial}{\partial x^m}(\mathbf{t}) = (\frac{\partial}{\partial x^m}(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * \Gamma^{p_1}_{m,i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * \Gamma^{p_r}_{m,i_r}$$

$$-t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * \Gamma^{j_1}_{m,q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * \Gamma^{j_s}_{m,q_s})*$$

$$(\frac{\partial}{\partial \mathbf{x^{p_1}}}, \ldots, \frac{\partial}{\partial \mathbf{x^{p_r}}}, \mathbf{dx^{q_1}}, \ldots, \mathbf{dx^{q_s}})$$

6.7 Kovariant derivasjon

Det er en nær sammenheng mellom kovariant derivasjon og partiell derivasjon. Vi vet hvordan vi skal beregne de partielle deriverte for deriverbare ringer og moduler, og kan derfor gi følgende definisjoner:

6.7.1 Kovariant derivasjon for ringer

Gitt en deriverbar ring R, sammen med modulen $D[R]$ av derivasjonsoperatorer for R. La $B = \{\frac{\partial}{\partial \mathbf{x^1}}, \ldots, \frac{\partial}{\partial \mathbf{x^d}}\}$ være basisen for $D[R]$. Vi definerer da den *kovariant deriverte* til et element $f \in R$ til å være et element $\mathbf{D_f} \in D[R]^*$, som har følgende egenskap :

$$< \mathbf{D_f}, \frac{\partial}{\partial \mathbf{x^i}} >= \frac{\partial}{\partial x^i}(f)$$

for $0 \leq i \leq d$. Vi sier da at $\mathbf{D_f}$ er den kovariant deriverte av f. Det kan lett vises at $\mathbf{D_f}$ kan representeres som følgende lineære kombinasjon:

$$\mathbf{D_f} = \frac{\partial}{\partial x^1}(f) * \mathbf{dx^1} + \ldots + \frac{\partial}{\partial x^d}(f) * \mathbf{dx^d}$$

der mengden $\{\mathbf{dx^1}, \ldots, \mathbf{dx^d}\}$ er basis for $D[R]^*$. Den kovariant deriverte til et ringelement f kalles ofte *gradienten* til f.

6.7.2 Kovariant derivasjon av moduler og tensorer

Gitt en deriverbar ring R, sammen med modulen $D[R]$ av derivasjonsoperatorer for R. La $B = \{\frac{\partial}{\partial \mathbf{x}^1}, \ldots, \frac{\partial}{\partial \mathbf{x}^d}\}$ være basisen for $D[R]$. Vi har videre gitt christoffelsymbolene $\Gamma^q_{i,j}$. Vi definerer kovariant derivasjon av deriverbare tensorer som en funksjon $covDiff : T^r_s(R) \rightarrow T^r_{s+1}(R)$, slik for $\forall \mathbf{t} \in T^r_s(R)$:

$$< covDiff(\mathbf{t}), \frac{\partial}{\partial \mathbf{x}^i} >= \frac{\partial}{\partial x^i}(\mathbf{t})$$

Dette gir oss at vi kan beregne den kovariant deriverte til en tensor $\mathbf{t} \in T^r_s(R)$:

$$covDiff(\mathbf{t}) = (\frac{\partial}{\partial x^m}(t^{p_1,\ldots,p_r}_{q_1,\ldots,q_s}) + t^{i_1,p_2,\ldots,p_r}_{q_1,\ldots,q_s} * \Gamma^{p_1}_{m,i_1} + \ldots + t^{p_1,\ldots,p_{r-1},i_r}_{q_1,\ldots,q_s} * \Gamma^{p_r}_{m,i_r}$$

$$-t^{p_1,\ldots,p_r}_{j_1,q_2,\ldots,q_s} * \Gamma^{j_1}_{m,q_1} - \ldots - t^{p_1,\ldots,p_r}_{q_1,\ldots,q_{s-1},j_s} * \Gamma^{j_s}_{m,q_s})*$$

$$(\frac{\partial}{\partial \mathbf{x}^{\mathbf{p_1}}}, \ldots, \frac{\partial}{\partial \mathbf{x}^{\mathbf{p_r}}}, \mathbf{dx}^{\mathbf{q_1}}, \ldots, \mathbf{dx}^{\mathbf{q_s}}, \mathbf{b}^{\mathbf{m}})$$

Merk at denne definisjonen og gir kovariant derivasjon for den deriverbare modulen $D[R]$ og den deriverbare komodulen $D[R]^*$.

6.8 Noen sammensatte derivasjonsoperatorer

Innenfor fysikken blir PDL'er ofte uttrykt ved hjelp av en del sammensatte derivasjonsoperatorer, de vanligste er *gradient* og *divergens*. Disse kan lett uttrykkes som en kombinasjon av de derivasjonsoperatorene vi allerede har definert for deriverbare tensorer.

6.8.1 Gradient

Gitt en deriverbar ring R, med d dimensjoner. Da er funksjonen:

$$grad : R \rightarrow D[R]^*$$

definert på følgende måte:

$$grad(f) = \frac{\partial}{\partial x^i}(f) * \mathbf{dx}^{\mathbf{i}}$$

Innenfor fysikk og mekanikk blir ofte *grad(f)* ofte benevnt som ∇f.

Merk her at gradienten til et ringelement f pr. definisjon er ekvivalent med den kovariant deriverte til f. Vi kan derfor gi en alternativ definisjon av *grad* som en derivasjonsoperator for tensorer:

$$grad : T_0^0(R) \rightarrow T_1^0(R)$$

slik:

$$\forall f \in T_0^0(R) : grad(f) = covDiff(f)$$

for en vilkårlig mengde av Christoffelsymboler Γ. Grunnen til at Γ her er et vilkårlig element, er at vi ikke trenger vite de partielle deriverte av basisen til $D[R]$ for å finne de partielle deriverte av et ringelement.

6.8.2 Divergens

Gitt en deriverbar ring R. Da er funksjonen:

$$div : T_s^r(R) \rightarrow T_{s-1}^r(R)$$

definert på følgende måte:

$$div(v) = Kontrakter(CovDiff(v))$$

Vi trenger her et christoffelsymbol Γ som angir hvordan de partielle av basisen til $D[R]$ er deriveres. Denne derivasjonsoperatoren kan implementeres mer effektivt enn det spesifikasjonen tilsier, siden operasjonen *kontrakter* ikke har bruk for alle de partielle deriverte som operasjonen *CovDiff* beregner.

6.9 Algebraisk spesifikasjon

Vi har sett at gitt en deriverbar ring R, sammen med mengden av alle derivasjonsoperatorer på R, kan vi definere derivasjonsoperatorer for den deriverbare modulen $D[R]$, den deriverbare komodulen $D[R]^*$ og det deriverbare tensorrommet $T_s^r(R)$. Dette kan vi benytte oss av for å spesifisere algebraisk hvilke egenskaper som må være gitt for disse spesielle sortene.

6.9.1 Deriverbare ringer

Vi kan først gi en algebraisk spesifikasjon av deriverbare ringer. Det er vanlige ringer som i tillegg har d derivasjonsoperatorer som er lineært uavhengige. Vi sier da at ringen har d dimensjoner:

```
specification DERIVERBAR-RING
include BOOL, HELTALL
include instantiation TABELL  by  diffRing
                using rTabell for Tabell
```

```
parameters
   sort        diffRing
   operations
      0                  :                           -> diffRing
      _ + _              :  diffRing * diffRing      -> diffRing
      - _                :  diffRing                 -> diffRing
      _ * _              :  diffRing * diffRing      -> diffRing
      numDim      ( _ )  :  diffRing                 -> Heltall
      partialDiff ( _,_ ) : Heltall * diffRing
                            | (0<#1<=numDim(#2))     -> diffRing
uses
   ikkeNullLoesn ( _ )   :  Heltall
                            | (0<#1<=numDim(0))      -> diffRing
   linSum      ( _,_,_ ) :  rTabell * diffRing* heltall
          | (lengde(#1)=numDim(#2)/\0<#3<=numDim(0)) -> diffRing
specifies
   variables   a, b : diffRing
               A    : rTabell
               i, j : Heltall
   equations
      //   partialDiff skal vaere derivasjonsoperatorer
      RING-DERIVASJON  ( ring, 0, 1, +, -, *, partialDiff(i,_) )

      //   spesifiserer dimensjonen til en ring. Ringen skal
      //   ha minst en dimensjon, og alle ringelementer skal
      //   ha like mange dimensjoner
      0          < numDim (b)
      numDim(a) == numDim (b)

      //   linSum beregner den lineaere kombinasjonen av tabellen A
      //   av ringelementer og de parielle deriverte til et ringelement b
      linSum(A,b,1) == A [ 1 ] * partialDiff( 1, b )
      linSum(A,b,i) == A [ i ] * partialDiff( i, b ) + linsum ( A, b, i-1 )

      //   Hvis linSum av tabellen A og 'ikkeNullLoesn(i)' er lik null, maa A kun
      //   maa A kun bestaa av nullelementer.  Det betyr at de partielle
      //   deriverte til ringen er lineaert uavhengige
      ( linSum ( A, ikkeNullLoesn(i), numDim(b) ) = 0 ) => ( A [ i ] = 0 )

      //   ikkeNullLoesn skal generere elementer som har egenskapen at
      //   de partielle deriverte av elementene ikke er null
      ! ( partialDiff ( i, ikkeNullLoesn (i) ) = 0 )
end specification
```

Her benytter jeg meg av den algebraiske spesifikasjonen RING-DERIVASJON som er gitt tidligere.

6.9.2 Christoffel

Siden vi ønsker å definere partiell derivasjon for en modul, trenger vi sorten *Christoffel*, som består av alle mulige christoffel-symboler. Christoffelsymbolene er indeksert v.h.a. tre heltall, og returnerer ring-elementer. Vi kan derfor gi følgende spesifikasjon:

```
specification CHRISTOFFEL
include HELTALL, BOOL
parameters
   sort       diffRing, Christoffel
   operations
      0                   :                      -> diffRing
      1                   :                      -> diffRing
      - _                 : diffRing             -> diffRing
      _ + _               : diffRing * diffRing  -> diffRing
      _ * _               : diffRing * diffRing  -> diffRing
      numDim      ( _ )   : diffRing             -> Heltall
      partialDiff ( _,_ ) : Heltall * diffRing
                            | (0<#1<=numDim(#2)) -> diffRing

      romDimensjon ( _ )  : Christoffel                      -> Heltall
      okIndeks ( _,_,_ )  : Heltall * Heltall * Heltall      -> Bool
      les ( _,_,_,_ )     : Christoffel * Heltall * Heltall
                            * Heltall | okIndeks(#2,#3,#4)   -> diffRing
      sett ( _,_,_,_,_ )  : Christoffel * diffRing * Heltall
                            * Heltall * Heltall
                            | okIndeks(#3,#4,#5)             -> Christoffel
specifies
   variables  d       : diffRing
              c       : Christoffel
              i, j, k,
              p, q, r : Heltall
   equations
      DERIVERBAR-RING ( diffRing, 0, 1, - , +, *,
                        numDim, partialDiff )

      romDimensjon ( c )   ==   numDim ( d )
      okIndeks ( i, j, k ) == (  0 < i,j,k  <= romDimensjon(c) )

      les ( sett ( c , d, i, j, k ), i, j ,k ) ==  d

      ( p != i /\ q != j /\ r != k ) =>
      les ( sett ( c , d, p, q, r ), i, j ,k ) == les ( c, i, j ,k ) )
end specification
```

Vi kan ikke si mye om egenskapene til sorten *Christoffel*, vi vet bare at vi kan indeksere dem med tre heltall som alle er i intervallet $[1, \ldots, d]$, der d er antallet dimensjoner for *diffRing*.

6.9.3 Deriverbare moduler

På samme måte kan vi fortsette for å spesifisere den deriverbare modulen $D[R]$ over en deriverbar ring R. Vi vet at $D[R]$ er en modul, og vi vet at basisen består av alle lineært uavhengige derivasjonsoperatorer for R. Dette betyr videre at modulen $D[R]$ har samme dimensjon som R.

```
specification DERIVERBAR-MODUL
include HELTALL, BOOL
parameters
   sort         diffRing, Christoffel, diffModul
   operations
      0                   :                         -> diffRing
      1                   :                         -> diffRing
      - _                 : diffRing                -> diffRing
      _ + _               : diffRing * diffRing     -> diffRing
      _ * _               : diffRing * diffRing     -> diffRing
      numDim      ( _ )   : diffRing                -> Heltall
      partialDiff ( _,_ ) : Heltall * diffRing
                            | (0<#1<=numDim(#2)) -> diffRing

      romDimensjon ( _ )    : Christoffel                          -> Heltall
      okIndeks ( _,_,_ )    : Heltall * Heltall * Heltall          -> Bool
      les ( _,_,_,_ )       : Christoffel * Heltall * Heltall
                              * Heltall | okIndks(#2,#3,#4)        -> diffRing
      sett ( _,_,_,_,_ )    : Christoffel * diffRing * Heltall
                              * Heltall * Heltall |okIndeks(#3,#4,#5) -> Christoffel

      0                 :                                          -> diffModul
      _ + _             : diffModul * diffModul                    -> diffModul
      - _               : diffModul                                -> diffModul
      _ * _             : diffRing * diffModul                     -> diffModul
      dimensjon ( _ )   : diffModul                                -> Heltall
      koordinat ( _,_ ) : diffModul * Heltall | (0<#2<=dimensjon(#1)) -> diffRing
      basis     ( _,_ ) : diffModul * Heltall | (0<#2<=dimensjon(#1)) -> diffModul

      partialDiff ( _,_,_ ) : Heltall * diffModul * Christoffel
                              Christoffel | (0<#1<=dimensjon(#2))    -> diffModul
      LieDiff ( _,_ )       : diffModul * diffRing                   -> diffRing
      LieDiff ( _,_ )       : diffModul * diffModul                  -> diffModul

uses
      Sum : Heltall * Heltall * Heltall * Christoffel  -> diffModul
specifies
   variables    v, w    : diffModul
                i, j, p : Heltall
                c       : Christoffel
                d, k    : diffRing
   equations
      CHRISTOFFEL      ( diffRing, Christoffel,
                         0, 1, - , +, *, numDim, partialDiff,
                         romDimensjon, okIndeks, les, sett   )

      DERIVERBAR-RING  ( diffRing,
                         0, 1, - , +, *, numDim, partialDiff  )
```

```
    RING-DERIVASJON  ( diffRing,
                       0, 1, - , +, *, numDim, LieDiff (v, _ ) )
    MODUL-DERIVASJON ( diffRing, diffModul,
                       0, 1, - , +, *, partialDiff (i, _ ),
                       0, -, +, *, dimensjon, koordinat, basis,partialDiff(i,_,c ) )
    MODUL-DERIVASJON ( diffRing, diffModul,
                       0, 1, - , +, *, LieDiff (v, _ ),
                       0, -, +, *, dimensjon, koordinat, basis, LieDiff (v, _ ) )

    //   fast dimension
    dimensjon ( v ) ==  numDim ( d )
    //   fast basis
    basis ( v, i )  ==  partialDiff ( i, _ )

    //   Lie derivert diffRing
    LieDiff ( basis ( v, i ), d )    == partialDiff ( i, d )
    LieDiff ( v + w         , d )    == LieDiff ( v, d ) + LieDiff ( w, d )
    LieDiff ( k * v         , d )    == k * LieDiff ( v, d )

    //   Lie derivert diffModul
    LieDiff ( LieDiff ( v, w ), d )  == LieDiff ( v, LieDiff ( w, d ) ) -
                                        LieDiff ( w, LieDiff ( v, d ) )

    //   partialDiff er en sum over basis og christoffel
    partialDiff ( i, basis(v,j), c ) == Sum ( numDim ( d ) ,i, j, c )

    //   den lokale operasjonen 'Sum' finner den partielle deriverte til
    //   basiselementene i modulen m.h.p. Christoffelsymbolet c
    Sum ( 1, i, j , c )              == les ( c, 1 , i, j ) * basis (v, 1 )
    Sum ( p, i, j , c )              == les ( c, p , i, j ) * basis (v, p ) +
                                        Sum (p-1,i,j,c )
end specification
```

Merk at den lie deriverte for deriverbare ringer er definert i denne spesifikasjonen. Det gjør vi siden vi er avhengig av denne spesifikasjonen for å gi aksiomene ang. modulelementet som er argument til *LieDiff*.

6.9.4 Deriverbare komoduler

Vi kan fortsette på samme måte for å spesifisere deriverbar komoduler og deriverbare tensorer. Siden vi nå har gitt spesifikasjonene DERIVERBAR-RING og DERIVERBAR-MODUL, vil dette være forholdsvis lett. Spesifikasjonen for deriverbare komoduler blir derfor:

```
specification DERIVERBAR-KOMODUL
include HELTALL, BOOL
parameters
   sort         diffRing, Christoffel, diffModul , diffKoModul
   operations
        :
```

```
    //   Her kommer alle operasjonene som er med i
    //   spesifikasjonen for DERIVERBAR-MODUL
       :

    //   diffKoModul operasjoner
    0                    :                                   -> diffKoModul
    _ + _                : diffKoModul * diffKoModul         -> diffKoModul
    - _                  : diffKoModul                       -> diffKoModul
    _ * _                : diffRing * diffKoModul            -> diffKoModul
    dimensjon ( _ )      : diffKoModul                       -> Heltall
    koordinat ( _,_ )    : diffKoModul * Heltall
                           | (0<#2<=dimensjon(#1))           -> diffRing
    basis     ( _,_ )    : diffKoModul * Heltall
                           | (0<#2<=dimensjon(#1))           -> diffKoModul
    _ ( _ )              : diffKoModul * diffModul           -> diffRing

    partialDiff ( _,_,_ ) : Heltall * diffKoModul *
                           Christoffel | (0<#1<=dimensjon(#2)) -> diffKoModul
    LieDiff ( _,_ )      : diffModul * diffKoModul           -> diffKoModul
specifies
   variables  c    : Christoffel
              m, n : diffModul
              v    : diffKoModul
              i    : Heltall
   equations
      DERIVERBAR-MODUL   ( diffRing, christoffel, diffModul,
                           0, 1, - , +, *, numDim, partialDiff,
                           romDimensjon, okIndeks, les, sett ,
                           0, -, +, *, dimensjon, koordinat , basis,
                           partialDiff, LieDiff, LieDiff          )

      KOMODUL-DERIVASJON ( diffRing, diffModul, diffKoModul,
                           0, 1, - , +, *,
                           partialDiff ( i, _ ),                     //   NB !
                           0, -, +, *, dimensjon, koordinat, basis,
                           partialDiff ( i, _, c ),                  //   NB !
                           0, -, +, *, dimensjon, koordinat, basis, _(_),
                           partialDiff ( i, _, c )                 ) //   NB !

      KOMODUL-DERIVASJON ( diffRing, diffModul, diffKoModul,
                           0, 1, - , +, *,
                           LieDiff ( m, _ ),                         //   NB !
                           0, -, +, *, dimensjon, koordinat, basis,
                           LieDiff ( m, _ ),                         //   NB !
                           0, -, +, *, dimensjon, koordinat, basis, _(_),
                           LieDiff ( m, _ )                        ) //   NB !
end specification
```

Merk at vi ikke trenger gi flere aksiomer enn de vi får fra spesifikasjonen KOMODUL-DERIVASJON. Det er siden en derivasjonsoperator for komoduler er entydig bestemt ut fra derivasjonsoperatorene til en ring og en modul. Det samme gjelder for spesifikasjonen

DERIVERBAR-TENSOR.

6.9.5 Deriverbare tensorer

spesifikasjonen for en deriverbar tensor blir:

```
specification DERIVERBAR-TENSOR
include HELTALL, BOOL
parameters
   sort      diffRing, Christoffel, diffModul, diffKoModul, diffTensor
   operations
         :
      //   Her kommer alle operasjonene som er med i
      //   spesifikasjonen for DERIVERBAR-KOMODUL
         :
         :
      //   Her kommer alle operasjonene som er med i
      //   spesifikasjonen for TENSOR
         :

      LieDiff ( _,_ )          : diffTensor * diffTensor | erModul ( #1 )  -> diffTensor
      partialDiff ( _,_,_ )    : Heltall * diffTensor *
                                 Christoffel |(0<#1<=numDim(0))            -> diffTensor
      covDiff ( _, _ )         : diffTensor * Christoffel                  -> diffTensor
      gradient ( _ )           : diffTensor              | erRing(#1)      -> diffTensor
      divergens ( _,_ )        : diffTensor * Christoffel
                                 | (0<antKomoduler(#1))                    -> diffTensor

specifies
   variables   t, t1     : diffTensor
               v         : diffModul
               c         : Christoffel
               i         : Heltall
   equations
      DERIVERBAR-KOMODUL    ( diffRing, christoffel, diffModul,
                              0, 1, - , +, *, numDim, partialDiff,
                              romDimensjon, okIndeks, les, sett ,
                              0, -, +, *, dimensjon, koordinat , basis,
                              partialDiff, LieDiff, LieDiff,
                              0, -, +, *, dimensjon, koordinat , basis, _ (_),
                              partialDiff, LieDiff                        )

      TENSOR-DERIVASJON     ( diffRing, diffModul, diffKoModul, diffTensor,
                              0, 1, - , +, *,
                              partialDiff ( i, _ ),                        //   NB !
                              0, -, +, *, dimensjon, koordinat, basis,
                              partialDiff ( i, _, c ),                     //   NB !
                              0, -, +, *, dimensjon, koordinat, basis, _(_),
                              partialDiff ( i, _, c ),                     //   NB !
                              ring, modul, komodul, antKomoduler, antModuler,
                              tensor, tensor, tensor, 0, +, -, *, dimensjon,
```

```
                                 koordinat, basis, *, <,>, kontraksjon, Kronecker,
                                 erRing, erModul, erKomodul,
                                 sammeType, gyldigIndreprod, kanKontraktere,
                                 partialDiff ( i, _, c )                       )  //  NB !

     TENSOR-DERIVASJON         ( diffRing, diffModul, diffKoModul, diffTensor,
                                 0, 1, - , +, *,
                                 LieDiff ( v, _ ),                                //  NB !
                                 0, -, +, *, dimensjon, koordinat, basis,
                                 LieDiff ( v, _ ),                                //  NB !
                                 0, -, +, *, dimensjon, koordinat, basis, _(_),
                                 LieDiff ( v, _ ),                                //  NB !
                                 ring, modul, komodul, antKomoduler, antModuler,
                                 tensor, tensor, tensor, 0, +, -, *, dimensjon,
                                 koordinat, basis, *, <,>, kontraksjon, Kronecker,
                                 erRing, erModul, erKomodul,
                                 sammeType, gyldigIndreprod, kanKontraktere,
                                 LieDiff ( tensor ( v ), _ )                   )  //  NB !

     < CovDiff ( t, c ), tensor ( basis ( v, i ) ) > == partialDiff ( i, t, c )

     gradient ( t )      == CovDiff ( t, c )
     divergens ( t, c ) == Kontrakter ( CovDiff ( t, c ) )
end specification
```

Merk at selv om *CovDiff* er definert for den deriverbare ringen R, modulen $D[R]$ og komodulen $D[R]^*$, blir den plassert sammen med operasjonene til $T^r_s(R)$, siden resultatet av operasjonen er en tensor av høyere rank.

Chapter 7

Implementasjon

7.1 Sorter som må implementeres

Det er ikke mulig å ha en representasjon av deriverbare ringer som er dekkende for alle formål og behov. Vi må implementere en rekke forskjellige abstrakte datatyper som er særlig egnet for spesielle oppgaver. Disse ADT'ene kan implementeres med forskjellige datastrukturer som gjør dem spesialiserte for spesielle felter. Noen eksemler kan være gitter i to og tre dimensjoner. Det kan og tenkes at en implementerer et gitter flere ganger med forskjellige derivasjonsoperatorer som kan variere med hensyn på nøyaktighet og tidsforbruk. Det eneste vi skal kreve av implementasjonene er at de oppfører seg som deriverbare ringer. Det gjør at vi kan implementere en parametrisert tensortype, som blir instantiert med den ringrepresentasjonen som vi selv ønsker. Vi har tidligere sett at alle deriverbar tensor operasjoner vi har spesifisert kan realiseres hvis vi har en datastruktur som oppfører seg som en deriverbar ring. Dette betyr at vi kan implementere en tensortype med alle operasjoner uten at vi binder oss til noen spesielle representasjoner! Hvis vi implementerer dette i et programmeringsspråk som har språkkonstruksjoner som støtter parametriserte datatyper, vil det være en fordel. Det vi da trenger å implementere er:

- En rekke forskjellige representasjoner av deriverbare ringer.
- En deriverbar tensor type som er parametrisert med en deriverbar ring. Det betyr at vi kan benytte samme tensorimplementasjon for alle mulige deriverbare ringer.
- En Christoffeltype som også er parametrisert med en deriverbar ring. Denne parametriserte typen trenger vi for å representere hvordan de partielle deriverte av basisen til $D[R]$ oppfører seg. Denne trenger vi heller ikke implementere mer enn en gang, siden den er parametrisert.

Siden vi , gitt en deriverbar ring, entydig kan konstruere tensorer av alle typer, trenger vi ikke lage egne implementasjoner av $D[R]$ og $D[R]^*$. Det er siden basiselementene til $T^r_s(R)$ er entydig gitt av den deriverbare ringen som er parameter til modulen. Vi har og vist at det er en isomorfisme mellom modulen $D[R]$ og $T^1_0(R)$, og mellom $D[R]^*$ og $T^0_1(R)$. Hvis vi har implementert $T^r_s(R)$, kan vi derfor fortsatt gjøre alle beregninger som vi kan ønske å gjøre med elementene i $D[R]$ og $D[R]^*$.

7.2 Valg av programmeringsspråk

Før vi velger språket vi ønsker å benytte under implementasjonen av de forskjellige modulene, må vi se mer på hvilke krav vi må stille til språket. Vi har følgende utgangspunkt:

1. Målet er å lage et verktøy som gjør det lettere å utvikle og vedlikholde programmer som behandler tensorfelt. For å oppnå dette definerer vi tensorfeltet som en abstrakt datatype med et entydig grensesnitt mot det kallende programmet.
2. En applikasjon som benytter seg av tensorklassen skal lett kunne kjøres på mange forskjellige maskinarkitekturer (både sekvensielle og parallelle maskiner).
3. Målgruppen til modulene som blir utviklet er programmerere som løser realistiske numeriske problemer relatert til tensorfelt.

Noen rimelige krav til språket som blir valgt vil derfor være :

1. Det må eksistere kompilatorer til språket på mange forskjellige maskiner. Evt. må språket være lett flyttbart.
2. Språket bør være effektivt til numeriske beregninger.
3. Det bør være kjent innen målgruppen, evt. bør det være lett å lære.
4. Språket bør støtte konstruksjon og bruk av abstrakte datatyper. Det er og viktig at en kan skrive generiske funksjoner og moduler (d.v.s. funksjoner og moduler som kan ta typer som argument).

Det finnes mange språk som støtter ett eller flere av disse kravene. De mest kjente språk som støtter flere av punktene over er ADA, ADTP, C++, ML og MODULA-2. Det som særlig kjennetegner disse språkene, er at de alle har innebygd språkkonstruksjoner som støtter utvikling og bruk av abstrakte datatyper.

Ada er utviklet av det amerikanske forsvar på slutten av 70-tallet. Det er et omfattende språk med konstruksjoner som støtter mange ulike retninger innen programmering. Noen eksempler er abstrakte datatyper og sanntid-programmering. Kompilatoren er forholdsvis omfattende. Syntaksen er ikke ulik Pascal, og er forholdsvis lett å ta i bruk. Språket er ikke særlig utbredt, men blir mer og mer benyttet på innen mange områder.

Adtp er utviklet ved UiB ca. 1988. Er en utvidelse av språket Pascal for å støtte konstruksjon og bruk av abstrakte datatyper og generiske moduler. Kompilatoren er på samme måte som C++ implementert som en preprosessor. Preprosessoren genererer kode i Pascal og kan derfra bli oversatt til C-kode. Språket er likevel ikke i dag tilgjengelig på mange arkitekturer. Det er siden Pascal ikke er særlig standardisert. I teorien er likevel språket like flyttbart som språket C. Adtp er lite kjent utenfor UiB, men er lett å ta i bruk.

C++ er en utvidelse av C for å støtte objektorientert systemutvikling. De siste utvidelser av standarden til språket inneholder både abstrakte datatyper og generiske funksjoner. Kompilatoren er oftest implementert som en preprosessor som genererer C-kode, og det gjør at C++ er et like portabelt språk som C. For programmerere som ikke allerede kan C er det forholdsvis høy brukerterskel for å beherske C++. Språket er allerede mye brukt, og bruken har raskt spredt seg til mange felt, også innen numerisk databehandling.

ML er et såkalt funksjonellt språk. Det vil si at et program i språket kun er bygd opp av funksjoner, det blir ikke benyttet globale variabler. Dette fører til at det heller ikke er tilordningssetninger i språket, og en er derfor garantert å unngå ringvirkninger fra funksjonene i programmet. Vi kan si at et program i ML beskriver mer hva vi ønsker å beregne enn hvordan vi skal beregne det. Dette gjør at språket er flyttbart til en mengde forskjellige datamaskiner, også parallelle. I ML er det lett å konstuere nye datatyper, og det er egen støtte for å lage abstrakte datatyper og generiske funksjoner. Dette gjør at språket er godt egnet for prototyping av nye datatyper. Siden språket er svært kraftig, har det vært et problem å utvikle kompilatorer som kan generere effektiv kode, særlig på parallelle maskiner. ML er lite utbredt utenfor universitetsmiljøet. Siden språket er svært ulikt andre konvensjonelle imperative språk, krever det en del arbeid å lære seg ML.

Modula-2 Dette språket er utviklet som en etterfølger av Pascal, og inneholder utvidelser for å støtte sanntids programmering og bruk av abstrakte datatyper. Språket er mye likt Pascal og er forholdsvis mye benyttet. Det finnes dessuten kompilatorer for en mengde forskjellige maskiner. Det er likevel ikke like utbredt som ADA, selv om det inneholder mange like konstruksjoner.

Disse språkene passer i varierende grad til våre ønsker, og enkelte av dem overlapper mye m.h.p. funksjonalitet. Ut fra våre behov kan vi trekke følgende konklusjoner :

ML :
- Det er for lite kjent innenfor numeriske miljør.
- For programmerere som er vant til imperative språk krever det mye arbeid å lære språket.
- Selv om det har en del teoretisk interessante egenskaper m.h.p. flytting mellom forskjellige parallelle arkitekturer, finnes det ikke i dag implementasjoner som egner seg for realistisk bruk.

Modula-2 : • For vårt bruk er ADTP mer anvendelig, siden ADTP har parametriserte typer. Dette har ikke Modula-2.

Ada : • Har mange konstruksjoner som er nyttige for vårt bruk.

• Er lett å ta i bruk p.g.a. likhet med mange andre mer kjente språk.

• Er bare delvis kjent innen numeriske miljøer.

Adtp : • Har de samme fordeler og ulemper som ADA for vårt bruk, men er i teorien lettere å flytte over til andre maskintyper.

• Er i dag ikke tilgjengelig på mange maskiner.

• Det er heller ikke særlig kjent utenfor UiB.

C++ : • Er mer kjent enn ADA og ADTP.

• Eksisterer kompilatorer for mange forskjellige maskiner.

• Inneholder de mange av de konstruksjonene som vi trenger.

• Har høyere brukerterskel enn ADA og ADTP.

• Kan generere svært effektiv kode.

Siden C++ er mest kjent, eksisterer på flere maskintyper, og støtter godt vårt behov vil antagelig dette språket være mest passende. Dette valget er ikke egentlig noen overraskelse, da det allerede er forholdsvis mye brukt innen numerisk databehandling. Det språket som enna er mest brukt er Fortran, men siden språket mangler elementære konstruksjoner for modulær oppbygging av abstrakte datatyper, er det flere og flere som finner at Fortran ikke er særlig egnet for store utviklingsprosjekter og gjenbruk av kode (se f.eks. [3] som inneholder en diskusjon om dette).

7.3 Tabeller

En tabell av elementer er samling med elementer av en bestemt type. Elementene i tabellen har ikke noen navn, den eneste måten en har tilgang på enkeltelementene i tabellen er ved referanse til en posisjon i tabellen. Det er svært avgjørende for effektiviteten til et program at implementasjonen av en tabell-struktur er effektiv. C++ har implementert en tabell-struktur som en del av språket, men implementasjonen har flere åpenbare svakheter:

- Antallet elementer i tabellene må være kjent når en kompilerer programmet.
- Det er ingen kontroll på om en indeks er innenfor grensene til en tabell.
- All referanse til en tabell-struktur skjer v.h.a. pekere. Det er lett å gjøre programmeringsfeil med pekere, og konsekvensene av feil bruk kan være alvorlige.

Det er derfor nyttig å implementere en egen tabell-stuktur som unngår svakhetene ved den innebygde konstruksjonen. Det er svært viktig at denne implementasjonen er effektiv. Grunnen til dette er at svært mange andre klasser vi skal lage består av tabeller. Mesteparten av den totale kjøretiden vil derfor bestå av tabell-operasjoner. Vi vil derfor få en relativt stor endring i ytelsen til et helt program hvis ytelsen til tabell-strukturen endrer seg. En tabell-operasjon som er særlig tidkrevende, er kopiering. Det er viktig at denne operasjonen er implementert effektivt siden C++ automatisk overfører argumenter til funksjoner, og resultatet fra funksjoner, ved hjelp av kopiering. En kan lett hindre kopieringer av argumenter til en funksjon ved kun å overføre referanser til argumentene (det er dette som vanligvis blir gjort i C++). Problemet er overføring av returverdien fra funksjoner. Hvis en funksjon returnerer en 'stor' datastruktur, vil overføringene kunne ta mye tid hvis vi bruker mange funksjonskall i et program (se [18] for en mer grundig diskusjon om dette). En vanlig måte å unngå dette på er å bruke en teknikk som kalles *referansetelling* (reference-counting, se [3] for en grundig innføring i denne teknikken). Denne teknikken er svært effektiv, og gjør at kopieringen kan skje i konstant tid. Siden tabell-strukturen er så viktig, skal vi gå raskt gjennom implementasjonen her.

Tabellklassen har følgende signatur:

```
template <class T> class array
{
  public:
      //  alle operasjoner etter noekkelordet 'public' er synlige
          for alle brukerene av klassen

      //  Konstruktorene til klassen. De har ikke noe funksjonsnavn
      //  i deklarasjonen, men kan kalles ved navnet 'array<T>'
inline array                (                       )       ;   //  1
inline array                ( const int    size     )       ;   //  2
       array                ( const int    size  ,
                              const T      & el     )       ;   //  3
inline array                ( const array & elArr   )       ;   //  4

      //  Denne funksjonen fjerner (deallokerer) en tabell
inline ~array               (                       )       ;   //  5

      //  Operatoren '==' tester for likhet mellom to tabeller
      int        operator  == ( const array & elArr  ) const ;  //  6

      //  Tilordningsoperatoren for tabeller
inline array &   operator  =  ( const array & elArr  )       ;  //  7

      //  Denne funksjonen returnerer lengden til en tabell
inline int       size         (                      ) const ;  //  8

      //  Operatoren '+' tar to tabeller som argument og
      //  returnerer en ny tabell som er konkateneringen av
      //  argumentene
```

```
      array      operator  +  ( const array &          ) const ;   //   9

      //   Oppslag og oppdatering av elementer i en tabell som
      //   ikke er deklarert som konstant
inline T &       operator  [] ( const int              )       ;   //   10

      //   Oppslag i tabeller som er deklarert som konstante
inline const T & operator  [] ( const int              ) const ;   //   11

   private:
      //   Alt som staar etter noekkelordet 'private' er usynlig
      //   for brukerene av klassen.  Her blir datastruktur
      //   og private hjelpefunksjoner deklarert.
      //   Datastrukturen skal vi komme tilbake til senere.
           :
           :
} ;
```

Vi deklarerer her klassen *array* til å være en klasse som parametriseres med klasseparameteret T. Dette er typen til elementene som tabellen skal inneholde. Alt som står etter nøkkelordet *private* er data og funksjoner som kun er tilgjengelig for funksjonene som tilhører klassen. Nøkkelordet *inline* foran enkelte funksjoner signaliserer at funksjonene skal ekspanderes ut i all kildekode som kaller de aktuelle funksjonene. Det gjør de funksjonene er særlig effektive å benytte, siden en unngår kostnaden med å aktivisere funksjonskall for å utføre funksjonene. Nøkkelordet *const* etter parameterlisten til enkelte funksjoner indikerer at de aktuelle funksjonene ikke oppdaterer objektet som kaller funksjonen. Merk og nøkkelordet '&' som angir at et parameter til funksjonen er en *referanse* til et objekt. Husk at en kan overlaste operatorene i C++ til og å gjelde for objektene i en klasse (f.eks. blir operatoren '+' her brukt for å konkatenere to tabeller).

C++ har en litt forvirrende syntaks, og det ligger svært mye implisitt informasjon her. Vi kan derfor se hvordan C++ signaturen vil se ut innenfor den formalismen vi tidligere har brukt. Semantikken er ikke så lett å uttrykke her, siden funksjoner i C++ kan oppdatere parametrene til en funksjon. Vi tar derfor bare med signaturdelen til spesifikasjonen:

```
sorts      T, array
operations

   //   Alle typer og klasser (inkludert T) vil implisitt
   //   i C++ ha foelgende operasjoner.  Dette er de eneste
   //   operasjonene arrayklassen forventer av klassen T.
   //   Disse operasjonene blir generert automatisk av kompilatoren,
   //   men de kan overlastes i en klasseimplementasjon.
   T ()                 :                              -> T
   T ( _ )              : const T &                    -> T
   _::~T ()             : T                            ->
   _ == _               : const T & * const T &        -> int
   _ = _                : T * const T &                -> T &
```

```
//   standardoperasjoner i arrayklassen
array<T> ()           :                                  -> array<T>      //  1
array<T> ( _ )        : const array<T> &                 -> array<T>      //  4
_::~array<T> ( _ )    : array<T>                         ->               //  5
_ == _                : const array<T> * const array<T> & -> int          //  6
_ = _                 : array<T> * const array<T> &      -> array<T> &    //  7

//   spesifikke operasjoner for arrayklassen
array<T> ( _ )        : int                              -> array<T>      //  2
array<T> ( _,_ )      : int * int                        -> array<T>      //  3
_.size ()             : const array<T>                   -> int           //  8
_ + _                 : const array<T> * const array<T> & -> array<T>     //  9
_ [ _ ] = _           : array<T> * int * T               -> T &           // 10
_ [ _ ]               : array<T> * int                   -> T &           // 10
_ [ _ ]               : const array<T> * int             -> const T &     // 11
```

Et enkelt eksempel på bruk av denne klassen er (vi refererer her til nummeret til den aktuelle funksjon som blir kalt i hvert uttrykk):

```
#include "array.h"

int main ( void )
{
   array<int>  intArr1              ,        \\  1
               intarr2 ( 2 )        ;        \\  2

   array<char> chrArr1 ( 10      ) ,         \\  2
               chrArr2 ( 10, 'a' ) ;         \\  3

   const array<char> chrArr3 ( 10, 'b' ) ;   \\  3

   char tmp ;

   tmp            = chrArr3 [ 0 ]     ;      \\  11
   chrArr1        = chrArr3           ;      \\  7
   chrArr2 [ 5 ]  = tmp               ;      \\  10
   intArr1        = intArr1 + intArr2 ;      \\  9, 7
}
```

Merk at vi lett kan instantiere klassen med både typene 'int' og 'char'. Det er heller ikke noe i veien for at vi kan kontruere tabeller av tabeller av f.eks. reelle tall. Det kan gjøres slik:

```
array<array<float> >  ArrArr ( 10, flArray ( 20, 12.34 ) ) ;
```

Her deklarerer vi en tabell som består av $10 * 20$ reelle tall som alle har verdien 12.34.

Klassen over er implementert ved hjelp av referansetelling. Det betyr at objektene i klassen kan kopieres i konstant tid. Dette gjøres i praksis ved å la flere objekter dele felles minne-lokasjoner hvis de inneholder felles data. En passende datastruktur for dette er:

```
  private:
    void   makeCopy  () ;

    int    arrSize      ;
    int *  refCnt       ;
    T   *  data         ;
} ;
```

der elementene har følgende funksjoner og datainvarianter:

- Pekeren *data* peker til en sekvens av elementer av typen *T*. Denne sekvensen er lagret som en sammenhengende minneblokk etter vanlig C og C++ tradisjon (f.eks. er sekvensen indeksert fra 0).
- Heltallet *arrSize* inneholder antallet elementer i tabellen. Dette betyr at sekvensen av elementer som *data* peker til inneholder *arrSize* elementer.
- Pekeren *refCnt* peker til et heltall som angir hvor mange objekter i klassen *array*<*T*> som deler felles minneblokk.
- Funksjonen *makeCopy* er en privat funksjon som allokerer en ny minneblokk, og lager en egen kopi av alle elementene i den felles minneblokken.

En må nå passe på at hvis ett objekt skal oppdatere sin datastuktur, må det først sikres at det ikke oppstår ringvirkninger til de andre objektene som deler felles minneblokk. Dette kan lettest illustreres ved følgende utsnitt av implementasjonen :

```
template <class T> inline
array<T>::array   ( const array<T> & a )
: arrSize ( a.arrSize ),
  data    ( a.data    ),
  refCnt  ( a.refCnt  )
{
      ++ ( * refCnt ) ;
}
```

Denne funksjonen er en konstruktør som lager en ny tabell som er en kopi av argumentet *a*. Det eneste som skjer i funksjonen er:

- Den nye tabellen får samme lengde som argumentet.
- Pekeren *data* blir satt til å peke til samme minne-lokasjon som argumentet.
- Pekeren *refCnt* blir satt til å peke til samme teller som argumentet.
- Til slutt blir innholdet av *refCnt* økt med en, slik at *refCnt* inneholder antallet objekter som deler felles minnelokasjon (vi forutsetter at *a.refCnt* var riktig satt før kopi-operasjonen).

Hvis ett av objektene nå ønsker å gjøre en oppdatering av minne-lokasjonen må vi passe på at vi ikke får ringvirkninger. Den eneste funksjonen i klassen som gjør oppdateringer, er funksjonen:

```
template <class T> inline
T &  array<T>::operator  [] ( const int i )
{
   if ( ( * refCnt ) != 1 )
      makeCopy () ;

   return  data[ i ] ;
}
```

Her sjekker vi først om det er kun er ett objekt som 'eier' minne-blokken som tabellen sitter i (da er *(* refCnt) == 1*). Hvis det er flere objekter som refererer til samme data, kalles funksjonen *makeCopy*, som oppretter en ny minneblokk som ingen andre objekter i klassen refererer til. Da vil det ikke skje noen ringvirkninger ved oppdateringen. Siden en stor del av koden i mange program består av å referere til elementer i tabeller, er det viktig at oppslag skjer effektivt. Den ene testen i funksjonen over kan derfor få et merkbart utslag for den totale kjøretiden til et program. Det er derfor en fordel og å implementere følgende funksjon :

```
template <class T> inline
const T & array<T>::operator  [] ( const int i ) const
{
   return  data[ i ] ;
}
```

Denne funksjonen fungerer kun som en observatør på elementene i klassen som er deklarert som konstante i et program (p.g.a. nøkkelordet *const* i signaturen). Vi trenger ikke her sjekke på antallet referanser til felles minnelokasjon, siden vi er garantert at det ikke skal skje noen oppdateringer på datastrukturen.

7.4 Deriverbare ringer

Vi fant i forrige kapittel at hvis vi har implementert en deriverbar ring i et programmeringsspråk, kan vi entydig implementere alle mulige deriverbare tensorer over ringen. Dette forutsetter at ringen har følgende operasjoner:

```
   sort Ring

   operations

      0               :                      -> Ring
      1               :                      -> Ring
      - _             : Ring                 -> Ring
      _ + _           : Ring * Ring          -> Ring
```

```
_ * _              : Ring * Ring          -> Ring
numDim ( _ )       : Ring                 -> Heltall
partDiff ( _,_ )   : Heltall * Ring
                      | ( 0<#1<=numDim(0) ) -> Ring
```

og at operasjonene tilfredstiller kravene i spesifikasjonen DERIVERBAR-RING (d.v.s. at implementasjonen er en modell av spesifikasjonen). Vi har til hensikt å implementere en tensorklasse ved hjelp av parametriserte typer, der den deriverbare ringen er et typeargument til tensoren. For at vi skal kunne implementere en slik tensorklasse i C++, må vi spesifisere signaturen til de klassene som er gyldige som typeargument. Vi må derfor bli enig om en signatur som skal være felles for alle implementasjoner av deriverbare ringer:

```
class diffRing
{
   public :

      //   Administrative funksjoner
       diffRing                  (                   )       ;
       diffRing                  ( const diffRing & )       ;
       ~diffRing                 (                   )       ;
       int         operator ==   ( const diffRing & ) const ;
       diffRing &  operator =    ( const diffRing & )       ;

       //   Grunnleggende deriverbar ring-funksjoner
       diffRing                  ( const int      & )       ;
       diffRing    operator  -   (                  ) const ;
       diffRing    operator  +   ( const diffRing & ) const ;
       diffRing    operator  *   ( const diffRing & ) const ;
static int         numDimensions (                  )       ;
       diffRing    partialDiff   ( const int        ) const ;

       //   Sammensatte funksjoner
       diffRing    operator  -   ( const diffRing & ) const ;
       diffRing &  operator +=   ( const diffRing & )       ;
       diffRing &  operator -=   ( const diffRing & )       ;
       diffRing &  operator *=   ( const diffRing & )       ;
} ;
```

Det er selvfølgelig ikke noe i veien for at enkelte implementasjoner av *diffRing* har flere funksjoner i tillegg, men de funksjonene vil aldri bli brukt av tensorklassen vi senere skal implementere.

De administrative funksjonene må vi ha med av programmeringstekniske hensyn. Disse funksjonene håndterer tilordning og tester på likhet mellom to objekter. I tillegg har vi alltid en destruktør (~diffRing ()) og en konstruktør som ikke tar noen argument (*default constructor*). Det siste er det alltid nyttig å ha med i en klasse, da innholdet til et objekt ikke alltid kan spesifiseres i samme øyeblikk som det blir deklarerert i et program.

Det er hensiktsmessig å benytte en konstruktør i stedet for konstant-funksjonene $0 :\longrightarrow diffRing$ og $1 :\longrightarrow diffRing$. Det er delvis siden vi i C++ ikke kan overlaste symbolene 0 og 1, og siden vi ofte er interessert i å konstruere objekter som er en sum av flere 1'ere (d.v.s. tallene 2,3,4, ...). Hvis vi vet at vi skal finne en sum av 1'ere, kan dette nesten alltid implementeres effektivt inne i klassen, men hvis en skal gjøre det i et kallende program må det gjøres eksplisitt med en rekke addisjoner med konstanten 1. Vi kan gi følgende algebraiske spesifikasjon av konstruktøren:

```
n : Heltall

( 0 <= n ) =>
diffRing ( n ) == if n = 0 then   ringNull
                         else   ringEn + diffRing ( n - 1 )
```

der *ringNull* er ringelementet 0 og *ringEn* er ringelementet 1.

For at klassen *diffRing* skal være en deriverbar ring slik vi har spesifisert tidligere, må vi nå kreve at klassen virkelig har de egenskapene vi forventer. Det kan vi gjøre ved å kreve at aksiomet DERIVERBAR-RING instantiert med operasjonene over er tilfredstilt:

```
DERIVERBAR-RING ( diffRing, diffRing ( 0 ), diffRing ( 1 ),
                  - , +, *, numDimensions, partialDiff       )
```

7.4.1 Returverdiproblemet

Et velkjent problem med C++ er noe som kalles returverdiproblemet ([18]). Det oppstår som en følge av at funksjonene i språket potensiellt kan returnere store datastrukturer. Dette er ikke egentlig noe problem i seg selv, men kan føre til at det blir utført mange unødvendige kopieringer. Dette kan best illustreres ved et programeksempel:

```
diffRing a, b, c, d, e ;

a = ( b + c ) * d + e ;
```

C++ er en preprosessor som genererer c-kode. Uten å være for presise, kan vi skissere hvordan kodelinjene over vil bli oversatt til c-kode:

```
diffRing_data a, b, c, d ;

defaultInit ( a ) ;
defaultInit ( b ) ;
defaultInit ( c ) ;
defaultInit ( d ) ;

{
   diffRing_data tmp1, tmp2, tmp3 ;
```

```
    defaultInit ( tmp1   ) ;
    defaultInit ( tmp2   ) ;
    defaultInit ( tmp3   ) ;

    add  ( tmp1, a   , b ) ;
    mult ( tmp2, tmp1, d ) ;
    add  ( tmp3, tmp2, e ) ;
    copy ( a   , tmp3    ) ;

    deallocate ( tmp1 ) ;
    deallocate ( tmp2 ) ;
    deallocate ( tmp3 ) ;
}
```

Vi ser her at C++ vil allokere tre temporære variabler som brukes til å mellomlagre deluttrykkene til $(b + c) * d + e$. Vanligvis vil objekter med diffRing-egenskaper bestå av svært store datastrukturer. Det fører til at C++ vil generere kode som bruker unødig mye plass, og det blir utført mange kopieringer mellom temporære variabler. For å unngå dette problemet må signaturen og inneholde følgende sammensatte operasjoner:

```
diffRing    operator -    ( const diffRing & ) const ;
diffRing &  operator +=   ( const diffRing & )       ;
diffRing &  operator -=   ( const diffRing & )       ;
diffRing &  operator *=   ( const diffRing & )       ;
```

Disse operasjonene er sammensatte, da de lett kan uttrykkes som en sammensetning av de grunnleggende ring-operasjonene:

```
( a - b  ) == ( a + ( - b ) )
( a += b ) == ( a = a + b   )
( a -= b ) == ( a = a - b   )
( a *= b ) == ( a = a * b   )
```

Ved hjelp av de sammensatte operasjonene kan vi nå skrive om koden over slik at den blir oversatt til mer effektiv c-kode:

```
diffRing a, b, c, d, e ;

a  = b ;
a += c ;
a *= d ;
a += e ;
```

C++ koden over vil bli oversatt til noe som dette:

```
diffRing_data a, b, c, d ;

defaultInit ( a ) ;
```

```
defaultInit ( b ) ;
defaultInit ( c ) ;
defaultInit ( d ) ;

copy        ( a , b ) ;
addEqual    ( a , c ) ;
multEqual   ( a , d ) ;
addEqual    ( a , e ) ;
```

Vi ser at vi da fullstendig unngår all allokering og deallokering av temporære variabler. I tillegg slipper vi en del parameteroverføringer i forbindelse med funksjonskall.

7.5 Gitter som deriverbar ring

Den absolutt mest vanlige typen deriverbar ring som blir brukt i forbindelse med tensorer er forskjellige gitterrepresentasjoner. De brukes vanligvis for å representere funksjoner fra d reelle tall til de reelle tall, eller mer generellt funksjoner som er av typen:

$$f : K^d \to K$$

der K er en normert kropp (D.v.s. en ring med divisjonsoperator og norm). Vanligvis er K mengden av de reelle eller komplekse tall. Et gitter er en diskretisering av domenet K^d til å være indeksert med d heltall med en øvre og nedre grense for heltallene i hver dimensjon. Vi kan lett implementere et slikt gitter på en slik måte at det får deriverbar-ring egenskaper. Et gitter i to dimensjoner kan implementeres med følgende signatur:

```
template <class Field, int y , int x> class grid2d
{
   public:
inline   grid2d                        (                  )       ;
inline   grid2d                        ( const grid2d   & )       ;
inline   ~grid2d                       (                  )       ;
inline   int          operator ==      ( const grid2d   & ) const ;
inline   grid2d &     operator  =      ( const grid2d   & )       ;

inline   grid2d                        ( const Field    & )       ;

inline   grid2d                        ( const int        )       ;
         grid2d       operator -       (                  ) const ;
         grid2d       operator +       ( const grid2d   & ) const ;
         grid2d       operator *       ( const grid2d   & ) const ;

         grid2d       operator -       ( const grid2d   & ) const ;
         grid2d &     operator +=      ( const grid2d   & )       ;
         grid2d &     operator -=      ( const grid2d   & )       ;
         grid2d &     operator *=      ( const grid2d   & )       ;

static
```

```
inline  int             numDimensions  (                      )        ;
        grid2d          partialDiff    ( const int dim        ) const  ;

inline  Field &         operator  ()   ( const int,const int)          ;
inline  const Field &   operator  ()   ( const int,const int) const    ;
} ;
```

Der klassen *Field* som er parameter til *grid2d* er typen til elementene som ligger i hver 'node'. Parameterene x og y angir her hvor mange punkter gitteret skal ha i hver dimensjon. Et objekt i klassen vil bestå av $x * y$ noder som alle inneholder et objekt fra klassen *Field*. Man kan indeksere elementene i gitteret ved hjelp av *operator ()*, som tar som argument to heltall, det ene i intervallet $[0, \ldots, y-1]$, og det andre i intervallet $[0, \ldots, x-1]$. En funksjon som representeres ved hjelp av et gitter må alltid ha et definisjonsområde som argumentene er gyldig innenfor. Dette definisjonsområdet kan variere fra funksjon til funksjon. For at klassen over skal være mest mulig anvendelig, kan vi sette gyldig definisjonsområde i intervallet $[Field(0), \ldots, Field(1)]$ for begge dimensjonene. Dette er ikke noen begrensing, siden vi alltid kan skalere argumentene til en vilkårlig funksjon til å ligge innenfor intervallet over (dette gjelder alltid for normerte kropper). En vanlig måte å beregne den partielle deriverte til et gitter i dimensjon 0 of 1 kan finnes ved hjelp av Taylor-rekker (der a er et gitter med $x * y$ noder):

$$a.partialDiff(0)(i,j) = \frac{a(i,j+1) - a(i,j-1)}{Field(x-1)}$$

$$a.partialDiff(1)(i,j) = \frac{a(i+1,j) - a(i-1,j)}{Field(y-1)}$$

for alle noder (i,j) der $0 \leq i < y$ og $0 \leq j < x$. Vi antar her at vi benytter *rundhopp* mellom dimensjonene. Det vil si at:

- $a(-1,j) == a(y-1,j)$
- $a(i,-1) == a(i,x-1)$
- $a(y,j) == a(0,j)$
- $a(i,x) == a(i,0)$

Merk at i denne definisjonen av partiell derivasjon trenger vi en konverteringsoperator fra *int* til *Field*, p.g.a. nevnerene $x-1$ og $y-1$. I C++ får vi denne konverteringsoperatoren automatisk fra konstruktøren '*Field (const int)*' som vi spesifiserte tidligere.

Det er vanlig å implementere et gitter som *grid2d* ved hjelp av en tabell med en eller to dimensjoner. Tabellklassen vi så på tidligere vil være et godt valg ved en implementasjon. Det er siden gitteret typisk består av mange elementer, og det kreves derfor mye arbeid å utføre en kopiering eller en parameteroverføring til en funksjon. Tabell-klassen klarer å utføre kopiering i konstant tid, og vil derfor være optimal med hensyn på dette. Den klart mest tidkrevende gitter-operasjonen vil vanligvis være partiell derivasjon, og hvis vi klarer å implementere den operasjonen effektivt, vil vi få en helt klar generell ytelsesforbedring. Noen viktige observasjoner er:

- Den partielle deriverte til konstante gitter (d.v.s. gitter som har samme verdi i alle noder) er alltid lik null.
- Gitter som har samme verdi i alle noder trenger ikke lagre $x * y$ objekter, det holder med ett objekt og et flagg som sier at feltet er konstant.
- Resultatet av en multiplikasjon mellom konstanten 0 og et annet ring-element er alltid konstanten null.
- Resultatet av en multiplikasjon mellom konstanten 1 og et annet ring-element er alltid lik ring-elementet.
- Resultatet av en addisjon mellom konstanten 0 og et annet ring-element er alltid lik ring-elementet.
- Koordinatene til mange tensorer består av mange null-element. Noen eksempler er den metriske tensoren i et kartesisk koordinatsystem og Hook's tensor for isotrope medier.

Disse observasjonene kan vi med hell benytte i en implementasjon av et gitter. En mulighet er å benytte følgende datastruktur:

```
private :

    int             isConstant ;
    Field           constVal   ;
    array<Field>    gridVal    ;
```

med følgende datainvarianter:

- *isConstant* er null hvis gitteret ikke er konstant, en ellers.
- Hvis *isConstant* er null er innholdet av *constVal* udefinert, ellers inneholder *constVal* verdien som ligger i alle noder i gitteret.
- Hvis *isConstant* er null, har *gridVal* ingen elementer (*gridVal.size () == 0*). Ellers har *gridVal* $x * y$ elementer, og innholdet til noden i punktet (i, j) ligger i *gridVal[i*x+j]*.

En implementasjon av operasjonen *partialDiff* kan nå sjekke om *isConstant* er lik en. Hvis den er det, kan den returnere et konstantgitter med verdien null i alle noder uten å gjøre noen beregninger, hvis ikke kan operasjonen fortsette med å beregne de partielle deriverte som vanlig. En algoritme for å utføre en multiplikasjon mellom to gitter kan være (vi bruker bare pseudo-kode):

```
grid2d<Field,y,x> multipliser ( const grid2d<Field,y,x> a,
                                const grid2d<Field,y,x> b  )
{
   if ( a.isConstant && b.isConstant )
      return grid2d<Field,y,x> ( a.constVal * b.constVal ) ;

   if ( a.isConstant ) {          //   Vi vet at b ikke er konst.

      if ( a.constVal == 0 )
         return grid2d<Field,y,x> ( 0 ) ;

      if ( a.constVal == 1 )
         return b ;

      < lag nytt gitter res, og alloker res.gridVal > ;
      < multipliser a.constVal med alle el i b.gridVal og
        lagre resultatet i res.gridVal > ;

      res.isConstant = 0 ;
      return res ;
   }

   if ( b.isConstant ) {          //   Vi vet at a ikke er konst.
      :
      //   Samme kode som i testen over, men bytt a og b
      :
   }

   //   Her vet vi at hverken a eller b er konstant

   < lag nytt gitter res, og alloker res.gridVal > ;
   const int max = x * y ;

   for ( register int i = 0 ; i < max ; ++ i ) {
      res.gridVal [ i ]  = a.gridVal [ i ] ;
      res.gridVal [ i ] *= b.gridVal [ i ] ;
   }
   res.isConstant = 0 ;

   return res ;
}
```

Samme type algoritmer kan brukes for de andre aritmetiske operasjonene til gitteret.

Det er implementert to versjoner av gitterklassen *grid2d*. Den ene bruker en tabell-representasjon uten å ta vare på informasjon om et felt er konstant, og den andre har implementert algoritmene over. Det er gjort noen tester på de to implementasjonene, og ved en simulering av bølger som følge av en eksplosjon, viste det seg at den vanlige implementasjonen jobbet i 10 timer, mens den versjonen som benytter algoritmene over brukte 2 timer. I tillegg brukte den siste versjonen mye mindre minne !

De to testene over benyttet seg begge av tensor-klassen som vi skal studere senere. Dette er et godt eksempel på at det lønner seg å bruke innkapsling av datastrukturer. Hvis tensor-klassen skulle ha full adgang til datastrukturen til en deriverbar ring, ville vi ikke kunne endre representasjonen av gitteret uten at vi samtidig måtte endre implementasjonen av tensorklassen. Et annet poeng er at både tabellklassen og gitterklassen har spesialiserte algoritmer for å øke ytelsen. Hvis vi skulle implementert referansetelling av tabeller direkte i gitterklassen, ville antagelig koden blitt så kompleks at det ville kreve svært mye arbeid å få en feilfri implementasjon.

7.6 Christoffel

Som nevnt tidligere, er det ikke mye man kan si om christoffel-symbolene som representerer den partielle deriverte til basiselementene i modulen $D[R]$, der R er en deriverbar ring. De er for oss kun en samling ringelementer som er indeksert ved hjelp av tre heltall, og har følgende mening:

$$\frac{\partial}{\partial x^i}\left(\frac{\partial}{\partial \mathbf{x^j}}\right) = \Gamma^q_{i,j} * \frac{\partial}{\partial \mathbf{x^q}}$$

Vi satte opp en algebraiske spesifikasjon i forrige kapittel som inneholder noen få operasjoner, og en implementasjon av tensor-klassen venter kun at disse operasjonene er tilgjengelig. Vi må likevel implementere noen flere funksjoner i en christoffelklasse, siden vi ikke har sagt noe om hvilke generatorer klassen skal ha. En implementasjon i C++ kan ha følgende signatur:

```
template <class diffRing> class christoffel
{
   public:

       christoffel                (                       )       ;
       christoffel                ( const christoffel &   )       ;
       ~christoffel               (                       )       ;
       int            operator == ( const christoffel &   ) const ;
       christoffel &  operator =  ( const christoffel &   )       ;

       //  Funksjonene under er ventet av tensorklassen
static int            spaceDimension (                    )       ;
static int            isProperIndex  ( const int, const int,
                                       const int          )       ;
       const diffRing & operator  () ( const int, const int,
                                       const int          ) const ;
       diffRing &       operator  () ( const int, const int,
                                       const int          )       ;
} ;
```

der funksjonen paa linje 17 brukes for å oppdatere elementene til et objekt i klassen. Merk at klassen tar som argument en klasse *diffRing*, der *diffRing* skal være en klasse med deriverbar ring egenskaper. En mulig datarepresentasjon for denne klassen er å benytte tabellklassen vi beskrev tidligere. Det gjør at objektene i denne klassen er effektive som argumenter ved funksjonskall:

```
private:
    array<diffRing> c ;
```

Vi har følgende abstraksjon ved denne implementasjonen:

- $c.size() == (spaceDimension())^3$
- $K(i,j,p) = K.c[(i * spaceDimension() + j) * spaceDimension() + p]$

der K er et element i klassen.Merk at vi følger vanlig C og C++ standard, og lar alle tabell-indekser ha null som nedre lovlige grenseverdi.

7.7 Tensorer

Vi ønsker nå å implementere en klasse som er en modell av den algebraiske spesifikasjonen for deriverbare tensorer. Som tidligere gjør vi det ved å ta utgangspunkt i vår algebraiske spesifikasjon, og 'oversetter' spesifikasjonen til en klassesignatur. Denne klassesignaturen kan vi så realisere med en passende datastruktur. Vi har som forutsetning at denne klassen er parametrisert med en deriverbar ring R. Siden den deriverbare ringen entydig definerer den deriverbare modulen $D[R]$ over R, trenger vi ikke ta som parameter til tensor-klassen en implementasjon av modulen $D[R]$. Grunnen er at siden $D[R]$ er entydig definert av R, trenger vi ikke implementasjonen av $D[R]$. Vi lar heller implementasjonen av $D[R]$ og $D[R]^*$ være et 'spesialtilfelle' av implementasjonen $T_s^r(R)$. Dette fører til at vi ikke trenger å implementere noen egne klasser for $D[R]$ og $D[R]^*$. Dette er en fordel, siden vi reduserer mulighetene for feil bruk av klassene som er utviklet, og implementasjonen av $T_s^r(R)$ blir lettere å ta i bruk.

Vi kan 'oversette' den algebraiske spesifikasjonen DERIVERBAR-TENSOR til følgende C++ signatur:

```
template <class diffRing> class tensor
{
   public:

       //  Administrative operasjoner
       tensor                       (                    )       ;
       tensor                       ( const tensor    &  )       ;
       ~tensor                      (                    )       ;
       int           operator ==    ( const tensor    &  ) const ;
```

```
       tensor  &        operator =    ( const tensor     &  )       ;

       //   observatorer
       int              isRing        (                     ) const ;
       int              isModule      (                     ) const ;
       int              isCoModule    (                     ) const ;
       int              equalType     ( const tensor     &  ) const ;
       int              canContract   (                     ) const ;
       int              numCoModules  (                     ) const ;
       int              numModules    (                     ) const ;

       //   Moduloperasjoner
       tensor           zero          ( const tensor     &  ) const ;
       tensor           operator   -  (                     ) const ;
       tensor           operator   +  ( const tensor     &  ) const ;
friend tensor<diffRing> operator   *  ( const diffRing   &,
                                   const tensor<diffRing> &  )      ;
       int              dimension     (                     ) const ;
       array<diffRing>  coordinates   (                     ) const ;
       array<tensor>    basis         (                     ) const ;

       //  rene tensoroperasjoner
       tensor                         ( const diffRing   &  )       ;
       operator         diffRing      (                     ) const ;
       tensor           operator   *  ( const tensor  &     ) const ;
       tensor           operator  ()  ( const tensor  &     ) const ;
       tensor           contract      (                     ) const ;
static tensor           kronecker     (                     )       ;

       //   derivasjonsoperatorer
       tensor           lieDiff       ( const tensor  &     ) const ;
       tensor           partialDiff   ( const int          ,
                                   const christoffel<diffRing> &  ) const ;
       tensor           covDiff
                                  ( const christoffel<diffRing> &  ) const ;

       //   Sammensatte operasjoner
       tensor           operator   -  ( const tensor     &  ) const ;
       tensor &         operator  -=  ( const tensor     &  )       ;
       tensor &         operator  +=  ( const tensor     &  )       ;
       tensor &         operator  *=  ( const diffRing   &  )       ;
} ;
```

Funksjonene over skal være en implementasjon av operasjonene som er gitt i den algebraiske spesifikasjonen, sammen noen funksjoner som er nødvendige av tekniske grunner (konstuktører og destruktør).

7.7.1 Noen nyttige konstruktører

I tillegg til operasjonene vi har i den algebraiske spesifikasjonen, er det nyttig å ha med noen generelle konstruktører:

```
tensor              ( const int, const int      ) ;
tensor              ( const int, const int ,
                      const array<diffRing>  &  ) ;
```

Disse funksjonene har følgende uformelle spesifikasjon:

```
numCoModules ( tensor ( r,s )) = r
numModules   ( tensor ( r,s )) = s
koord ( tensor ( r,s )) = array<diffGrid> ( i, diffGrid(0) )

numCoModules ( tensor ( r,s,a ) ) = r
numModules   ( tensor ( r,s,a ) ) = s
koord ( tensor ( r,s,a ))         = a
```

for alle $r, s \geq 0$, $d = \mathit{spaceDimension}\ ()$, $i = d^{r+s}$ og a er en tabell av ring-elementer med lengde i.

7.7.2 Implementasjon av tensor-klassen

Etter vår spesifikasjon, og etter begrensningen vi har nevnt i tidligere, ser vi at alle tensorer av samme type og har samme basis. Siden denne basisen kun er synlig innenfor tensor-klassen, og er entydig gitt fra den deriverbare ringen som tensorene er instantiert med, trenger vi ikke ha noen spesiell lagringsstruktur som inneholder basis-elementene til en tensor, vi trenger kun lagre koordinatene. Vi vet at en tensor av type (r, s) har d^{r+s} koordinater, der d er antallet dimensjoner til den deriverbare ringen, og vi har tidligere sett at alle tensoroperasjoner kan uttrykkes ved operasjoner på koordinat-representasjonen. Vi kan derfor benytte følgende lagringsstruktur for tensorer:

```
private:
    array<diffRing> arr ;
    int             r   ,
                    s   ,
                    d   ;
```

der vi har følgende invarianter for en tensor $t \in T^r_s(R)$:

- $\mathit{arr.size()} = d^{r+s}$.
- d er lik antallet dimensjoner til den deriverbare ringen. Dette tallet trenger egentlig ikke lagres, da det kan finnes gjennom funksjonen *diffRing::numDimensions()*. Det kan likevel lønne seg å lagre det, da det blir svært mye brukt i klassen.
- r representerer antallet komoduler en tensor kan ta indreprodukt over.

- s representerer antallet moduler en tensor kan ta indreprodukt over.
- Koordinatet $t^{i_1,\ldots,i_r}_{i_{r+1},\ldots,i_{r+s}}$ er lagret i tabellen *arr* i posisjon $\sum_{j=1}^{r+s}(i_j * d^{j-1})]$.

Med representasjonen over er det lett å implementere de vanligste konstruktørene. Noen eksempler er:

```
template <class diffRing> inline
tensor<diffRing>::tensor  ( const tensor<diffRing> &  v )
: d   ( v.d   ),
  r   ( v.r   ),
  s   ( v.s   ),
  arr ( v.arr )
{}

template < class diffRing > inline
tensor<diffRing>::tensor (  const   int                 rr    ,
                            const   int                 ss    ,
                            const   array<diffRing> & mArr      )
: d   ( diffRing::numDimensions () ),
  r   ( rr                         ),
  s   ( ss                         ),
  arr ( mArr                       )
{
   if ( arr.size () != pow ( d, r + s ) )
      exit ( -1 ) ;
}
```

Implementasjonen av de vanligste observatørene er like lett å implementere:

```
template <class diffRing>  inline
int tensor<diffRing>::isRing (    ) const
{
   return ( r + s == 0 ? 1 : 0 ) ;
}

template < class diffRing > inline
tensor<diffRing>::operator diffRing    ( ) const
{
   if ( ! isRing () )
      exit ( -1 ) ;

   return arr [ 0 ] ;
}
```

Det kreves mer arbeid for å implementere de mer vanlige tensor-operasjonene. Grunnen til dette er at det lett blir mange indekser man må holde oversikt over og oppdatere når man f.eks. adderer to tensorer eller beregner et indreprodukt. Vi har i tidligere kapitler vist hvordan koordinat- representasjonen til tensorene endrer seg som en følge av operasjonene vi ønsker å utføre på tensorene. Alle operasjonene kan derfor uttrykkes ved hjelp av koordinat-representasjon og en manipulering av indekser. Ett eksempel på dette er:

$$x = y + z \Rightarrow x^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} = y^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} + z^{i_1,\ldots,i_r}_{j_1,\ldots,j_s}$$

Vi skal snart se på en implementasjon av denne operasjonen.

7.7.3 En nyttig indeks-klasse

Uttrykket over skal implisitt evalueres for alle indeks-verdier. Det vil lette arbeidet med implementasjonen av denne tensor-klassen hvis vi kan finne en god metode å representere og bearbeide indekser. En god måte å gjøre dette på, er å implementere en egen indeks-klasse som en abstrakt datatype. Hvis vi gjør dette vil vi lett kunne implementere de aller fleste tensor-operasjonene på en intuitiv måte. Det vi har behov for er:

- En representasjon av indekser som er en sekvens av heltall.
- Hvert heltall i sekvensen kan være i intervallet 1 til d, der d er øvre grense for alle heltall i sekvensen.
- For å beregne indreprodukt og tensorprodukt, er det nyttig å konkatenere to indekser.
- Vi ønsker å kunne utføre en beregning for alle mulige verdier av en indeks. Det bør derfor være lett å kunne 'løpe' over alle indeksverdier.

Det letter implementasjonen av tensorklassen mye hvis vi har en slik ADT tilgjengelig, og vi bør derfor utvikle en datatype med disse egenskapene. Vi se først på klassesignaturen til ADT'en, og beskriver operasjonene uformellt. Etterpå gir vi en algebraisk spesifikasjon. Signaturen er:

```
class index {
  public:
    //  Standard operasjoner, standard semantikk
    index               (                 )       ;
    ~index              (                 )       ;
    int     operator == ( const index &   ) const ;
    index & operator =  ( const index &   )       ;

    //  Operasjonene til klassen
    index               ( const int len,
                          const int max   )       ;
    index   operator +  ( const index& i  ) const ;
    void    operator ++ (                 )       ;
    void    reset       (                 )       ;
    void    set         ( const int pos,
                          const int val   )       ;

    int     overflow    (                 ) const ;
```

```
    int     pos            (                ) const ;
    int     size           (                ) const ;
    int     upperLimit     (                ) const ;
    int     read           ( const int pos  ) const ;
    int     mayIncrement   (                ) const ;
    int     maxPosition    (                ) const ;
} ;
```

Følgende punkter kan gi en intuitiv forklaring av de viktigste funksjonene:

- For alle $0 \leq x, y$, vil konstruktøren *index(x, y)* opprette en indeks som består av x heltall, som alle har lovlige verdier i intervallet $0, \ldots, y-1$. Alle heltallene får verdien null.
- Operatoren '+' konkatenerer to indekser med lik øvre grense.
- Funksjonen *reset* nullstiller alle heltallene i en indeks.
- Vi kan endre verdien til et heltall i en indeks med operasjonen *set*. Etter vanlig C++ tradisjon starter alle indekser fra null, og *pos* kan derfor være i intervallet null til lengden minus en. *val* må være mellom null og øvre grense for indeks-sekvensen.
- Funksjonen *size* returnerer antallet heltall som en indeks inneholder.
- Vi finner øvre grense for heltallene i en indeks med funksjonen *upperLimit*. Sagt på en annen måte: *index(x, y).upperLimit () == y*.
- For å lese verdien til ett av heltallene til en indeks kan vi bruke observatøren *read*.
- Funksjonen *pos* benyttes for å beregne et heltall etter følgende formel: *i.pos () =* $\sum_{j=0}^{i.size()-1}$ *i.read(j)*pow(i.upperLimit(),i.size()-1-j)*. Denne funksjonen er svært nyttig, da vi nå kan benytte en indeks til å posisjonere oss i en endimensjonal tabell.
- Operatoren '++' vil oppdatere heltallene i en indeks i, slik at følgende krav er oppfyllt: *i.pos() + 1 = (++ i).pos()*. En intuitiv forklaring til denne funksjonen er at funksjonen prøver å øke siste heltallet i en indeks i med en. Hvis heltallet da blir større enn den øvre grensen, blir heltallet satt lik null, og funksjonen vil prøve å inkrementere det nest siste tallet. Dette vil funksjonen eventuellt fortsette med til den har inkrementert et heltall, eller alle heltallene i sekvensen har nådd den øvre grensen. Hvis vi ikke kan inkrementere noen heltall (d.v.s. at alle heltall i indeksen er ved sin øvre grense), settes et flagg som sier at indeksen har overflyt. Verdien til dette flagget kan observeres med funksjonen *overflow*. Hvis en indeks har fått overflyt, returnerer funksjonen et tall ulik null.

Vi kan nå gi en algebraisk spesifikasjon. For å unngå problemet med å spesifisere semantikken til funksjonene som oppdaterer argumentene (f.eks. operasjonene '++' og *set*) vil jeg i denne spesifikasjonen anta at oppdater-operasjonene returner elementet som blir oppdatert:

```
specification INDEKS
include INTEGER
parameters
   sort     index
   operations
      index  ( _,_ )         : int * int | ( 0 <= #1 /\ 0 < #2 )            -> index
      _.reset ()             : index                                        -> index
      _ + _                  : index  * index
                               | (#1.upperLimit() == #2.upperLimit())       -> index
      ++ _                   : index          | #1.mayIncrement()           -> index
      _.set  ( _,_ )         : index * int * int
                               | ( 0<#2<size(#1) /\ 0<=#3<#1.upperLimit() )  -> Indeks

      _.overflow      ()     : index                                        -> int
      _.size          ()     : index                                        -> int
      _.upperLimit    ()     : index                                        -> int
      _.read          (_)    : index * int | ( 0<= #2 < #1.size() )          -> int
      _.pos           ()     : index                                        -> int
      _.mayIncrement  ()     : index                                        -> int
      _.maxPos        ()     : index                                        -> int
uses
      pow (_,_)              : int * int                                    -> int
      _ / _                  : int * int | ( #2 <> 0 )                      -> int
      _ MOD _                : int * int | ( #2 <> 0 )                      -> int
specifies
   variables   i, j    : Indeks
               p, q, r : Heltall
   equations
      index ( p, q ).overflow () == 0
      i.reset().overflow ()      == 0
      ( i + j ).overflow ()      == i.overflow () + j.overflow ()
      i.set(p,q).overflow ()     == i.overflow ()
      (++i).overflow ()          == ( i.pos () == i.maxPos ()

      index(p,q).size ()         == p
      i.reset ().size ()         == i.size ()
      ( i + j ).size ()          == i.size () + j.size ()
      (++i).size ()              == i.size ()
      i.set(p,q).size ()         == i.size ()

      index(p,q).upperLimit ()   == q
      i.reset().upperLimit ()    == i.upperLimit ()
      ( i + j ).upperLimit ()    == i.upperLimit ()
      ( ++ i ).upperLimit  ()    == i.upperLimit ()
      i.set(p,q).upperLimit ()   == i.upperLimit ()

      i.maxPos ()                == pow ( i.upperLimit (), i.size () )

      index(p,q).pos ()          == 0
      i.reset().pos()            == 0
      ( i + j ).pos ()           == i.pos () *
```

```
                                pow ( j.upperLimit(), j.size () ) + j.pos ()
    ( ++i ).pos ()           == i.pos () + 1
    i.set(p,q).pos ()        == i.pos () + ( q - i.read(p) ) *
                                pow ( i.upperLimit(), i.size() -r )

    i.mayIncrement ()        == ( ! i.overflow () + i.pos () <= i.maxPos () )

    i.read ( r )             == ( i.pos() MOD pow(i.upperLimit(),i.size()-r) ) /
                                  pow ( i.upperLimit(), i.size() - r )

    pow  ( 0, p )            == 1
    pow  ( p, 0 )            == 1
    pow  ( p, q )            == p * pow ( p, q - 1 )
    p MOD q                  == p - ( p / q ) * q
    ( p  < q ) => ( p / q )  == 0
    ( q <= p ) => ( p / q )  == ( ( p - q ) / q ) + 1
end specification
```

Vi må anta at spesifikasjonen INTEGER inneholder en spesifikasjon av de vanligste operasjonene til typen *int* i C++. Det er flere måter å man kan velge ved en implementasjon av klassen *index*. En mulghet er å benytte følgende datarepresentasjon:

```
private:
   int          max      ;
   int          overflow ;
   array<int>   intArr   ;
```

der elementene i strukturen har følgende mening:

- Heltallet *max* angir øvre grense for alle heltallene i indeksen.
- Vi markerer eventuellt overflyt fra operatoren '++' med å sette *overflow* til et heltall ulik null. Hvis indeksen ikke har overflyt inneholder *overflow* tallet null.
- Tabellen *intArr* inneholder alle heltallene som indeksen består av.

Vi kan som et eksempel på bruk av denne klassen, vise en mulig implementasjon av operatoren '+' mellom to tensorer av samme type (r, s). Vi vet at vi kan beregne summen av to tensorer y og z av type (r, s) på følgende måte:

$$x^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} = y^{i_1,\ldots,i_r}_{j_1,\ldots,j_s} + z^{i_1,\ldots,i_r}_{j_1,\ldots,j_s}$$

Addisjonen over skal gjøres for alle mulige indekser til en tensor. En implementasjon som benytter seg av indeks-klassen er følgende:

```
template <class diffRing>
tensor<diffRing>  tensor<diffRing>::operator  +
                     ( const tensor<diffRing> & v ) const
{
   if ( ! equalType ( v ) )
```

```
      exit ( -1 ) ;

   tensor<diffRing> res ( * this   ) ;
   index            i   ( r + s, d ) ;

   while ( ! i.overflow () ) {
      res.arr[ i.pos () ] += v.arr [ i.pos () ] ;
      ++ i ;
   }
   return res ;
}
```

Dette er ikke den mest effektive måten å implementere funksjonen over, siden denne algoritmen ikke krever noen avanserte permutasjoner av indeksene til en tensor. Algoritmen gir likevel et godt innblikk i bruk av klassen *index*. Implementasjonen under nok være mye mer effektiv for akkurat dette eksempelet:

```
template <class diffRing>
tensor<diffRing>  tensor<diffRing>::operator  +
                      ( const tensor<diffRing> & v ) const
{
   if ( ! equalType ( v ) )
      exit ( -1 ) ;

   tensor<diffRing> res ( * this    ) ;
   const int        max = arr.size () ;

   for ( register int i = 0 ; i < max ; ++ i )
      res.arr[ i ] += v.arr [ i ] ;
   }
   return res ;
}
```

Det neste eksempelet fra implementasjonen krever mye mer avansert håndtering av indekser, og da vil indeksklassen vise seg å være mye mer nyttig.

7.7.4 Implementasjon av en viktig tensor-operasjon

Det er særlig tre tensor-operasjoner som kan være litt problematiske å implementere. Det er *tensorprodukt*, *indreprodukt* og *kontraksjon*. Grunnen til dette er at det i disse funksjonene er svært mange indekser som man må manipulere. Ved hjelp av indeks-klassen kan vi likevel konstruere algoritmer for disse operasjonene som er oversiktlige og enkle. Som et eksempel på hvordan en kan implementere disse operasjonene kan vi se hvordan vi kan implementere operasjonen *tensorprodukt*.

Tensorprodukt

Operasjonen *tensorprodukt* er en funksjon som tar som argument to tensorer, og returnerer en ny tensor. Hvis vi har argumentene $t_1 \in T^{r_1}_{s_1}(R)$ og $t_2 \in T^{r_2}_{s_2}(R)$ vil vi få som resultat en tensor $t_3 \in T^{r_1+r_2}_{s_1+s_2}(R)$. Vi har tidligere utledet at koordinat-representasjonen til t_3 kan beregnes på følgende måte:

$$(t_3)^{i_1,\ldots,i_{r_1},m_1,\ldots,m_{r_2}}_{j_1,\ldots,j_{s_1},n_1,\ldots,n_{s_2}} = (t_1)^{i_1,\ldots,i_{r_1}}_{j_1,\ldots,j_{s_1}} * (t_2)^{m_1,\ldots,m_{r_2}}_{n_1,\ldots,n_{s_2}}$$

(summert over alle like indekser).Vi kan av uttrykket se at vi har fire forskjellige indekssekvenser vi må forholde oss til. Det er sekvensene "$i_1,\ldots,i_{r_1}$", "$m_1,\ldots,m_{r_2}$", "$j_1,\ldots,j_{s_1}$" og "$n_1,\ldots,n_{s_2}$". Vi skal beregne alle mulige koordinat-verdier for t_3, og må derfor gjøre beregningingene for alle mulige kombinasjoner av indeksene. En grovalgoritme blir derfor:

```
Tensorprodukt :

inn :   t1 : Tensor av type ( r1, s1 )
        t2 : Tensor av type ( r2, s2 )

ut  :   t3 : Tensor av type ( r1 + r2, s1 + s2 )

start:

   A : Indeks med r1 heltall
   B : Indeks med s1 heltall
   C : Indeks med r2 heltall
   D : Indeks med s3 heltall

   For alle kombinasjoner av A
      For alle kombinasjoner av B
         For alle kombinasjoner av C
            For alle kombinasjoner av D
               t3(A,C,B,D) = t1(A,B) * t2(C,D) ;
            sluttFor ;
         sluttFor ;
      sluttFor ;
   sluttFor ;
slutt ;
```

Ved hjelp av indeks-klassen vår kan vi forholdsvis lett overføre algoritmen over til en implementasjon i C++:

```
template <class diffRing>
tensor<diffRing> tensor<diffRing>::operator * ( const tensor<diffRing> & t2 ) const
{
   index              A    ( r    , d ) ;
   index              B    ( s    , d ) ;
   index              C    ( t2.r, d ) ;
   index              D    ( t2.s, d ) ;
   tensor<diffRing>   t3   ( r + t2.r, s + t2.s ) ;

   while ( ! A.overflow () ) {
      B.reset () ;
      while ( ! B.overflow () ) {
```

```
        C.reset () ;
        while ( ! C.overflow () ) {
           D.reset () ;
           while ( ! D.overflow () ) {
              t3.arr[(A+C+B+D).pos()]  = arr[(A+B).pos()] * t2.arr[(C+D).pos()] ;
              ++ D ;
           }
           ++ C ;
        }
        ++ B ;
     }
     ++ A ;
  }
  return t3 ;
}
```

Dette er en oversiktlig implementasjon, men den er ikke særlig effektiv. Grunnen til dette er alle konkateneringene av indekser. Slik som indeks-klassen er implementert, vil det derfor bli mye allokering og deallokering av tabeller. Det er ikke noe i veien for at en kan lage en ny implementasjon av indeks-klassen som er mer effektiv ved konkatenering (det kan gjøres ved å alltid allokere større tabeller enn en trenger, og så konkatenere ved å kopiere inn elementene). En må likevel ikke glemme hvilke operasjoner som er mest kostbare i implementasjonen over, det er antageligvis operasjonene som behandler objekter i klassen *diffRing*. Hvis f.eks. *diffRing* er et gitter med 400 * 400 reelle tall, vil en multiplikasjon i koden over bestå av 160000 multiplikasjoner mellom reelle tall (i tillegg kommer kostnadene ved eventuelle kopieringer av argumenter og resultat). Dette fører til at kostnaden som er forbundet med manipulering av indekser ofte er forsvinnende liten i forhold til kostnaden til en multiplikasjon. En kan derfor vanligvis ikke forvente noen særlig gevinst i kjøretid ved å optimalisere indeks-operasjonene. Hvis derimot kostnaden ved en multiplikasjon er forholdsvis liten, vil mesteparten av kostnadene komme fra manipulering av indekser. Dette kan f.eks. intreffe hvis klassen *diffRing* er et gitter som er implementert på en parallell maskin med like mange prosessorer som gitteret har noder. En multiplikasjon mellom to gitter kan da skje svært effektivt.

Siden vi vet at indeks-klassen vår ikke er effektiv ved konkatenering, kan vi lage en ny implementasjonen av algoritmen over til å ta hensyn til dette.Forskjellen mellom denne implementsjonen og den forrige, er at her bruker vi ikke konkatenering av indekser i det hele tatt. Vi oppretter heller tre ekstra indekser som skal inneholde samme verdier som konkateneringene ville hatt. I tillegg ser vi at indeks-verdien til *arr* er gitt allerede i den andre while-løkken. Vi trenger derfor ikke beregne dette i den innerste løkken. Implementasjonen blir defor:

```
template <class diffRing>
tensor<diffRing> tensor<diffRing>::operator * ( const tensor<diffRing>  & t2 ) const
{
   register int     i                                 ;
```

```
   int               resPos                               ;
   int               arg1pos                              ;
   int               arg2pos                              ;
   index             A         ( r                  , d ) ;
   index             B         ( s                  , d ) ;
   index             C         ( t2.r               , d ) ;
   index             D         ( t2.s               , d ) ;
   index             resIndx   ( r + t2.r + s + t2.s, d ) ;
   index             arg1Indx  ( r + s              , d ) ;
   index             arg2Indx  ( t2.r + t2.s        , d ) ;
   tensor<diffRing> t3         ( r + t2.r, s + t2.s     ) ;

   while ( ! A.overflow () ) {
      B.reset () ;
      for ( i = 0 ; i < r ; ++ i ) {
         arg1Indx.set ( i, A.read ( i ) ) ;
         resIndx.set  ( i, A.read ( i ) ) ;
      }
      while ( ! B.overflow () ) {
         C.reset () ;
         for ( i = 0 ; i < s ; ++ i ) {
            resIndx.set  ( i + r + t.r, B.read ( i ) ) ;
            arg1Indx.set ( i + r, B.read ( i ) ) ;
         }
         arg1pos = arg1Indx.pos () ;
         while ( ! C.overflow () ) {
            D.reset () ;
            for ( i = 0 ; i < t.r ; ++ i ) {
               resIndx.set ( i + r, C.read ( i ) ) ;
               arg2Indx.set ( i, C.read ( i ) ) ;
            }
            while ( ! D.overflow () ) {
               for ( i = 0 ; i < t.s ; ++ i ) {
                  resIndx.set  ( i + r + t.r + s, D.read ( i ) ) ;
                  arg2Indx.set ( i + t.r, D.read ( i ) ) ;
               }
               arg2pos = arg2Indx.pos () ;
               resPos  = resIndx.pos  () ;
               t3.arr [ resPos ]  = arr    [ arg1pos ] ;
               t3.arr [ resPos ] *= t2.arr [ arg2pos ] ;
               ++ D ;
            }
            ++ B ;
         }
         ++ C ;
      }
      ++ A ;
   }
   return t3 ;
}
```

7.8 Implementerte klasser

For tiden er det implementert seks forskjellige klasser. Det er klassene:

- array
- christoffel
- grid2d
- index
- matrix
- tensor

Vi kan kort beskrive egenskapene til klassene slik:

Array : Dette er en implementasjon av en C-lik tabelltype som er parametrisert med typen til elementene som skal være i tabellen. Klassen er implementert med referansetelling, og det gjør den spesiellt egnet til store tabeller med mye kopiering. Det blir benyttet så mye 'inlining' som mulig. Denne klassen tillater tabeller som inneholder null elementer. Det er lagt mye arbeid i å gjøre denne klassen effektiv. Klassen inneholder ikke egen minne-håndterer, så en kan kanskje forbedre ytelsen noe ved å lage spesialiserte versjoner av *new* og *delete*. Disse operasjonene bør i tilfelle prøve å redusere antall kall til *malloc* og *free* ved å bygge sin egen liste eller søketre av ledige minneblokker, Disse abstrakte datatypene kan selvfølgelig ikke være implementert ved dynamiske strukturer.

Christoffel : Denne klassen inneholder informasjon om hvordan basisen til et rom med d dimensjoner deriveres. Klassen *tensor* er avhengig av denne klassen for å utføre partiell derivasjon. Siden christoffel-symboler ikke er tensorer (symbolene er ikke invariante ved skifte av basis), kan vi betrakte denne klassen som en tredimensjonal tabell av ring-elementer. Klassen er implementert som beskrevet foran. Det vil si at vi benytter tabellklassen over for å lagre elementene til et christoffelsymbol.

grid2d : Dette er en klasse som kan brukes for å representere et gitter. Klassen tar som parameter antall punkter i x- og y-retning, typen til elementene som skal ligge i alle nodene i gitteret, og et heltall, n, som angir hvor nøyaktig de partielle deriverte skal beregnes. Typen til node-elementene må ha egenskaper tilsvarende en *kropp*, og vanligvis vil dette være de innebygde typene *float* og *real*. Det er heller ikke noe i veien for å intantiere klassen med komplekse tall hvis en har en slik abstrakt datatype tilgjengelig. Klassen har og implementert divisjonsoperator. Heltallet n angir hvor mange noder i hver retning klassen skal benytte for å utføre en partiell derivasjon. Dette betyr at klassen har implementert en $2n$ punkts derivasjonsoperator som klassen beregnet v.h.a. Taylorrekke approksimasjon. For at beregningen av de partielle deriverte skal være så effektiv som mulig, benytter klassen rundhopp. Klassen er og optimalisert med hensyn på å utføre binære operasjoner og partiell derivasjon på konstantfelter. Dette gjør at klassen er særlig egnet for bruk sammen med tensorklassen. Eksperimentering har vist at derivasjonsoperatoren vil gi en liten feil hvis et gitter inneholder bølger, og $1 < n$. For å gjøre denne klassen mer passende for representasjon av bølger, vil klassen hvis $n = 7$, benytte en alternativ derivasjonsalgoritme som er tilfredstillende nøyaktig for bølger (klassen benytter da 14 punkts *Holberg* derivasjonsoperator).

index : Dette er en implementasjon av klassen *index*, og brukes i implementasjonen av klassen *tensor* for å lettere håndtere indeksene til tensorene.

matrix : Denne klassen inneholder en del matriseoperasjoner som vi trenger for å implementere klassen *grid2d* og klassen *tensor*. Siden denne klassen bare brukes for å lett implementere andre klasser, er det ikke gjort mye arbeid for å implementere denne klassen som en generell matrise-klasse. Klassen inneholder derfor kun de operasjonene som er nødvendige for de andre klassene. Det er operasjoner for å beregne *determinanter* og den *inverse* av en matrise.

tensor : Denne klassen er en implementasjon av deriverbare tensorer. Den er implementert med alle de operasjoner som er nevnt foran, og benytter seg av samme implementasjonsstrategi som tidligere er skissert. Selv om denne klassen kun tar som klasseparameter en deriverbar ring, er den implementert på en slik måte at det er lett å reimplementere den til og å ta som klasseparameter en deriverbar modul. Dette vil gi oss den fordelen at klassen da også kan anvendes på *generelle mangfoldigheter*. En kan anslå omskrivingsarbeidet for tensorklassen til ca. to dagsverk.

Totalt består klassene av ca. 3600 linjer kode, og klassene *grid2d* og *tensor* inneholder ca. 2400 av disse linjene.

Chapter 8

Eksempel : Seismisk modellering

Geofysikk er studiet av bevegelser og aktiviteter i deler av jorden. En del av geofysikk er seismikk, som er studiet av bølgeforplantning i jorden. Seismisk modellering er en viktig del av seismikk. Ved å simulere hvordan bølger brer seg i et medium, kan en finne nyttig informasjon som gir en bedre forståelse av seismikk generellt. Vi vil her implementere et enkelt program som beregner hvordan bølger vil oppstå og spre seg som en følge av en eksplosjon i et medium. Siden vi her kun skal vise hvordan tensor-klassen vår kan brukes, forutsetter vi at en del av den matematiske bakgrunn er kjent. Mer detaljer om elastitets-likninger på tensorform finner man i [12]. En kan finne en mye mer grundig innføring i seismisk modellering i [14], og mange detaljer i dette kapittelet om numerisk stabilitet kommer derfra.

Det er en del viktige begreper og sammenhenger innen dette området:

Metrikk : Et medium som bølger beveger seg gjennom, har et lengdemål som er representert ved hjelp av den metriske tensoren g. Den metriske tensoren er av type $(0,2)$. Vi har og en invers metrisk tensor g^*, som er av type $(2,0)$.

Stress : *Stress* er en tensor av type $(2,0)$. Den kan fysisk tolkes som spenningen som virker på et enhetselement i modellen som vi skal simulere over.

Strain : Når et element blir påført *stress*, vil elementet bli deformert. Deformasjonen av et element benevnes som *strain*, og er en tensor av type $(0,2)$.

Hook's lov : Når et element blir påført deformasjon, er spenningen på elementet direkte proporsjonalt med deformasjonen. Denne lineære kombinasjonen kan uttrykkes med en tensor *hook*, av type $(4,0)$. Vi har da sammenhengen:

$$stress = hook(strain)$$

Det vil si at stresset kan finnes ved å ta et indreprodukt mellom *hook* og *strain*.

Forflytning : Når et medium er deformert, kan deformasjonen uttrykkes som en tensor u, av type $(1,0)$, som angir forflytningen til alle enhetselement i mediet fra likevektsposisjon. Ved hjelp av den metriske tensoren og forflytningen, kan vi beregne mediets *spenning* på følgende måte:

$$strain = g.lieDiff(u)$$

Tetthet : Hvert enhetselement i et medium har en masse. Den uttrykkes vanligvis som vekt pr. enhetselement, og er en tensor av type $(0,0)$.

Ytre påkjenning : Når en skal simulere hvordan bølger brer seg, trenger man en eksplosjonskilde. Denne kilden påfører mediet en kraft som videre gir opphav til en deformasjon. For at eksplosjonen skal kunne uttrykkes uavhengig av metrikken til rommet, er det vanlig at eksplosjonen blir uttykt på følgende form:

$$expl = defVal * g^*$$

der *defVal* er en tensor av type $(0,0)$, og g^* er den inverse metriske tensoren. *expl* er derfor en tensor av type $(2,0)$, og blir lagt til mediets em stress. Vi får derfor at stresset blir:

$$stress = hook(strain) + defVal * g^*$$

Akselerasjon : Hvis et medium ikke er i likevekt, vil deformasjonen til mediet endre seg som en funksjon av tiden. Denne endringen kommer av kreftene som virker på mediet, og gir opphav til en akselerasjon på alle enhetselementer. Gitt stress og en metrisk tensor, kan vi beregne akselerasjonen på følgende måte:

$$accl = invDiff * stress.divergence(c)$$

Der c er et christoffel-symbol som er beregnet fra g og g^*, og *invDiff* er den inverse tettheten til mediet. Vi ser derfor at akselerasjonen er en tensor av type (1, 0).

Tidsutvikling : Fra fysikkens lover vet vi at akselerasjon kan uttrykkes ved hjelp av forflytning som en differensiallikning:

$$\frac{\partial^2(accl)}{\partial^2 t} = u$$

Dette betyr at gitt en akselerasjon, kan vi beregne forflytningen. For å gjøre dette, trenger vi en numerisk metode som gir oss en mulighet til å utføre en integrasjon av *accl* m.h.p. tiden. En vanlig algoritme for dette er:

$$u_{t+dt} = 2 * u_t - u_{t-dt} + dt^2 * accl_t$$

der dt er en liten endring i tid, u_{t+dt} er forflytningen vi ønsker å beregne for tidspunktet $t + dt$, u_t er forflytningen ved tiden t, og u_{t-dt} er forflyningen ved tiden $t - dt$.

Starttilstand : Når en skal beregne forflytningen som en funksjon av tiden v.h.a. likningen over, trenger en å vite tilstanden (forflytningen) til mediet ved tidspunktet $t = -dt$, og $t = 0$. En svært vanlig starttilstand er at mediet ikke har noen deformasjon eller hastighet ved tidspunktet $t = 0$. Dette betyr at både u_0 og u_{-dt} er nulltensorer.

Ved hjelp av opplysningene over kan vi sette opp følgende grovalgoritme for en simulering av bølger i et medium (som en følge av en ytre påkjenning):

```
inn : antSkritt : Heltall
      dt        : ring
      invDens   : diffRing
      g         : Tensor
      g_dual    : Tensor
      hook      : Tensor
      c         : christoffel

begin
     var  u0,u1, tmp, strain,stress, accl : Tensor

     u0     = nullTensor ( 1, 0 )  ;
     u1     = nullTensor ( 1, 0 )  ;

     for i = 0 to antSkritt
        strain = g.lieDiff ( u0 ) ;
        stress = hook ( strain ) +
                 g_dual * defVal ( ring(i) * dt );
        accl   = invDens * stress.divergence ( c ) ;
        tmp    = u1 + u1 - u0 + dt * dt * accl ;
        u0     = u1 ;
        u1     = tmp ;
     endfor

     return u1 ;
end
```

Denne algoritmen er helt uavhengig av det underliggende koordinatsystem. Det eneste som kreves er at koordinatsystemet er *metrisk* (d.v.s. at det finnes en metrisk tensor som kan beskrive avstandsmålet til koordinatsystemet). Merk at denne algoritmen kaller opp en funksjon $defVal$, som har følgende signatur: `diffRing defVal ( const ring &`

`time ) ;`. Det er denne funksjonen som beregner den ytre kraften som virker på mediet vi simulerer over.

Ved hjelp av gitterklassen vår kan vi nå implementere en versjon av algoritmen over. Gitterklassen brukes til å representere rom med to dimensjoner, og vi kan derfor konstruere en todimensjonal modell av algoritmen. Vi kan da bruke følgende konfigurasjon av klassene:

```
typedef   float                     ring        ;
typedef   grid2d<ring,N,SIXE,SIZE>  diffRing    ;
typedef   tensor<diffRing>          Tensor      ;
typedef   christoffel<diffRing>     Christoffel ;
```

der N er et heltall som angir hvor nøyaktig derivasjonsoperator gitterklassen skal benytte, og $SIZE$ angir antallet noder gitteret har i hver dimensjon. Vi vet at vi skal simulere hvordan bølger brer seg, og det er derfor nyttig å velge $N = 7$, siden gitter-klassen da vil benytte en derivasjonsoperator som er særlig nøyaktig for bølger. Det er i grunn vilkårlig hvor stort område vi ønsker å simulere over, men p.g.a. feilkildene ved beregningen av de partielle deriverte i gitterklassen, er det viktig at utstrekningen av området står i forhold til antallet noder vi bruker for å representere gitteret. Vi kan uttrykke utstrekningen som $RANGE$, der $RANGE$ angir antall km. mediet har i hver dimensjon.

8.1 Oppsett av metrikk

Vi vil i resten av vårt eksempel anta at vårt domene er et vanlig kartesisk koordinatsystem i to dimensjoner, og at rommet har utstrekningen $RANGE$ i hver dimensjon. Dette betyr at den metriske tensoren i vårt domene er den vanlige:

$$g_{i,j} = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

Vi må likevel ikke glemme at gitter-klassen vår har utstrekning $[0.0, \ldots, 1.0]$ i hver dimensjon. Dette betyr at vi ikke kan bruke g direkte som vår metriske tensor under simuleringen, vi må beregne en ny metrisk tensor, g', som angir metrikken til gitteret. Hvis vi angir aksene til vårt opprinnelige domene som x og y, og aksene til gitteret som x' og y', kan vi sette opp følgende sammenheng:

$$x' = \frac{1}{RANGE} * x + 0 * y$$

$$y' = 0 * x + \frac{1}{RANGE} * y$$

merk her at:

$$\begin{vmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} \end{vmatrix} = \begin{vmatrix} \frac{1}{RANGE} & 0 \\ 0 & \frac{1}{RANGE} \end{vmatrix}$$

Dette betyr at basisen i vårt koordinatsystem har følgende transformasjonsmatrise:

$$A = \begin{vmatrix} \frac{1}{RANGE} & 0 \\ 0 & \frac{1}{RANGE} \end{vmatrix}$$

Vi vet fra tidligere at når basisen har A som transformasjonsmatrise, vil koordinatene transformeres etter matrisen A^{-1}. Vi vet at:

$$A^{-1} = \begin{vmatrix} RANGE & 0 \\ 0 & RANGE \end{vmatrix} = \begin{vmatrix} \frac{\partial x}{\partial x'} & \frac{\partial x}{\partial y'} \\ \frac{\partial y}{\partial x'} & \frac{\partial y}{\partial y'} \end{vmatrix}$$

og at koordinatene til g transformerer på følgende måte:

$$g'_{ij} = g_{pq} * (A^{-1})^p_i * (A^{-1})^q_j$$

Dette betyr at koordinatene til g' blir:

$$g'_{ij} = \begin{vmatrix} RANGE^2 & 0 \\ 0 & RANGE^2 \end{vmatrix}$$

I C++ kan dette uttrykkes på følgende måte:

```
Tensor g = Tensor::identityTensor(0,2) * diffRing ( RANGE * RANGE ) ;
```

8.2 Beregning av Hook's tensor

Hook's tensor er av type $(4,0)$, og inneholder sammenhengen mellom *strain* og *stress* for et medium. En kan si at denne tensoren sier noe om *elastiteten* til et medium. Vi vil i dette eksempelet begrense oss til såkalte *isotrope* medier. Det vil si at de elastiske egenskapene til mediet ikke varierer i de forskjellige dimensjonene til mediet. Dette gjør vi siden det da er særlig lett å beregne verdiene til tensoren. De fleste bergarter er dessuten tilnærmet isotrope, og dette er derfor en rimelig forutsetning. Det kan vises at koordinatene til hook's tensor da blir:

$$hook^{ijpq} = \mu * g *^{ip} *g *^{qj} +0.5 * \lambda * g *^{ij} *g*^{pq}$$

der μ og λ er tensorer av type (0,0) og kalles Lame's konstanter. Disse konstantene kan beregnes hvis vi har gitt følgende materialkonstanter for vårt medium:

Tettheten : Dette er samme tetthet som vi beskrev tidligere.

Vp : Partikkelhastigheten til et enhetselement i retning normalt med retningen til en bølge.

Vs : Partikkelhastigheten til et enhetselement i retning ortogonormalt til bølgeretningen.

Lame's konstanter kan beregnes etter følgende formler:

$$\mu = Vs^2 * density$$

$$\lambda = Vp^2 * density - 2 * my$$

Oversatt til C++ får vi derfor dette uttrykket for hook's tensor:

```
const diffRing my     = vs * vs * density ;
const diffRing lambda = vp * vp * density - my - my ;

hook  = ( my * gInv * gInv ).swapUpperindex ( 1, 2 ) +
        ptFive * lambda * gInv * gInv               ;
```

8.3 Beregning av ytre kraft

Kilden til bølgene i vårt eksempel kan plasseres i en eller flere noder i vårt gitter. Av numeriske grunner er det en fordel at kilden plasseres i mer enn ett punkt. Selve verdien til nodene i kilden bør være en 'glatt' funksjon m.h.p. tidsutviklingen (d.v.s. at den tidsderiverte til kilden bør være forholds liten). Dette og av numeriske grunner. En bølgekilde som er egnet, er noe som kalles en Ricker bølgefunksjon, og denne bølgen har følgende form:

$$wave(t, f) = (1 - 2(\pi * f * t)^2)e^{-(\pi * f * t)^2}$$

der t er tiden og f er frekvensen til bølgen som inneholder mest bevegelsesenergi.

8.4 Stabilitet

Det er flere ting man må ta hensyn til når en skal modellere kontinuerlige funksjoner på et diskretisert gitter. Det kan vises at følgende betingelser må være oppfyllt for at den numeriske løsning til vår simulering skal være stabil:

1. $max(Vp) * \frac{dt}{min(dx)} < \frac{1}{\sqrt{2}}$
2. $max(dx) < \frac{min(Vs)}{2*f}$

der $max(Vp)$ er den maksimale partikkelhastighet i modellen, $min(dx)$ er den minimale avstand mellom to noder i gitteret, dt er tidsintervallene vi integrerer over, $max(dx)$ er maksimal avstand mellom to noder, $min(Vs)$ er minimal partikkelhastighet og f er maksimal frekvens for bølgen som blir lagt til modellen. Det kan vises at den maksimale frekvensen kan være opp til dobbelt så høy som frekvensen til den mest energirike frekvensen. Betingelse nr. 1 sier at det må være et visst forhold mellom antall noder vi har i gitteret og hvor store tidsskritt vi kan benytte i simuleringen. Betingelse nr. 2

sier hvor høy frekvens vi kan ha i vår bølgekilde i forhold til antallet noder i gitteret (vi trenger et visst antall noder for å representere en hel periode av en bølge). Vi ser at gitt gitterstørrelsen, Vs og Vp, kan vi finne en passende frekvens f og en passende dt etter formlene over:

$$f < \frac{min(Vs) * SIZE}{4 * RANGE}$$

$$dt < \frac{RANGE}{\sqrt{2} * max(Vp) * SIZE}$$

8.5 Simulering av en modell

En vanlig oppgave innen seismisk modellering er å simulere hvordan en bølge vil spre seg som en følge av at den treffer en overgang mellom to bergarter med forskjellig fysiske egenskaper. Dette kan vi og simulere med modellen vår over. Vi kan plassere en eksplosjonskilde midt i gitteret, og legge en overgang mellom to bergarter et sted mellom kilden og randen. Dette gjøres ved å initiere gitterene Vs, Vp og $density$ med forskjellige verdier over og under overgangen mellom bergartene. Selve implementasjonen av en slik modell er svært lett når en først kjenner den fysiske modellen og har passende abstrakte datatyper tilgjengelig (les: gitterklasse og tensorklasse). Implementasjonen er på ca. 350 linjer, og er lagt ved som et vedlegg. Modellen baserer seg på et gitter med $400 * 400$ noder, og bruker en 14 punkts Holberg derivasjonsoperator ved beregningen av de partielle deriverte. Det tar ca. fire timer å kjøre dette programmet på en SUN SPARCstation 10, og resultatet kan illustreres med de vedlagte figurene (visualisert med MATLAB). Resultatet fra simuleringen er en datafil som inneholder $400 * 400$ reelle tall, og resultatet kan ikke få plass i denne teksten med full oppløsning. Resultatet vises derfor i redusert format der vi plukker ut hver fjerde node i hver dimensjon, og får derfor bare presentert resultatet med en oppløsning på $100 * 100$ noder. Dette fører til at figurene ikke er like nøyaktige som resultatet i datafilen.

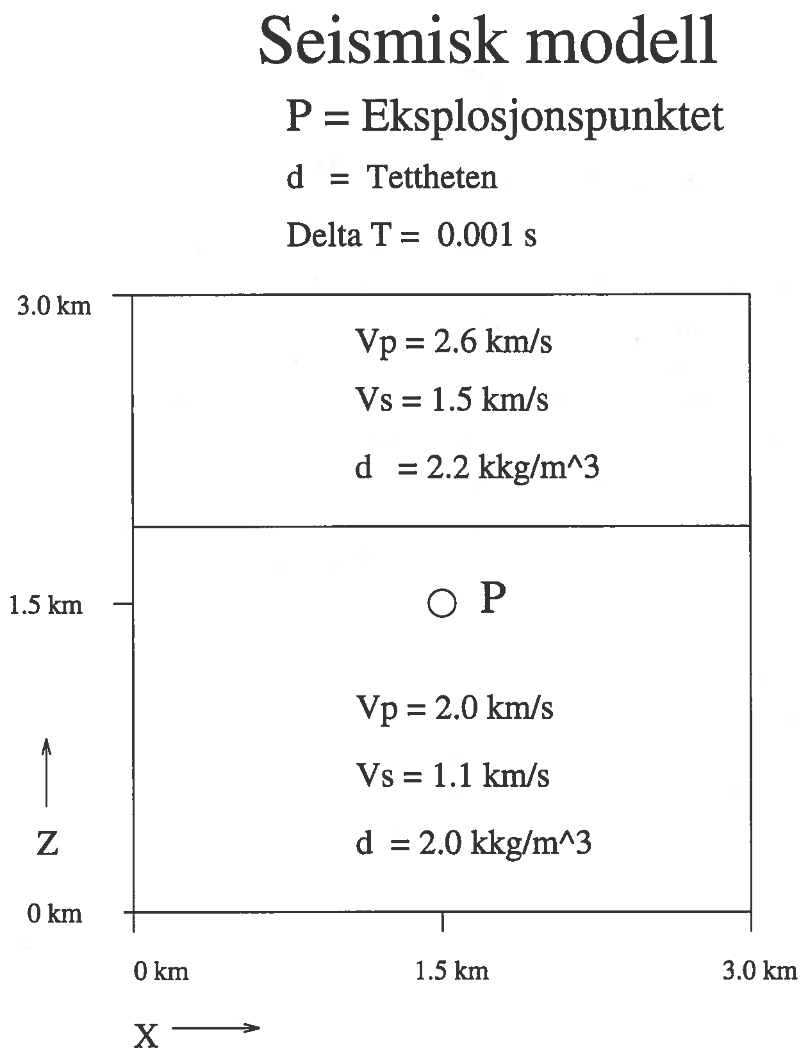

Figure 8.1: Simuleringsoppsett.

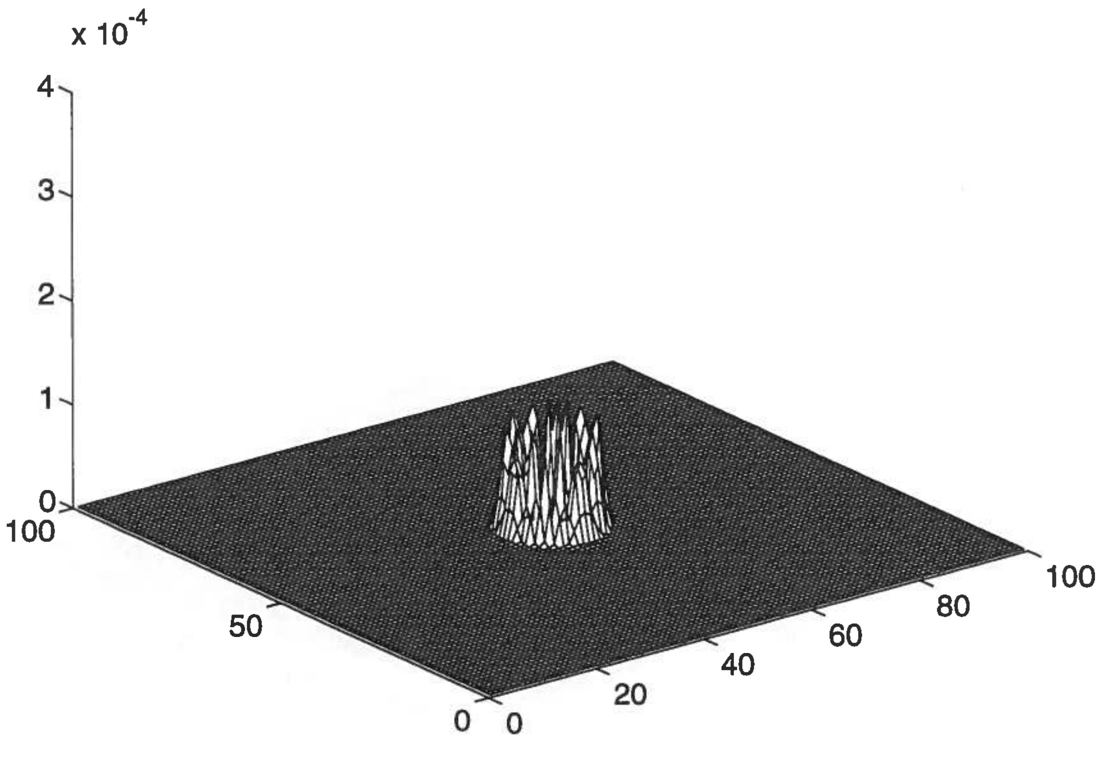

Figure 8.2: Resultat etter 150 iterasjoner. Vi ser her forflytningen (amplituden) bort fra likevektsposisjon for alle enhetselementer i modellen. Skalaen for forflytningen er i *km*.

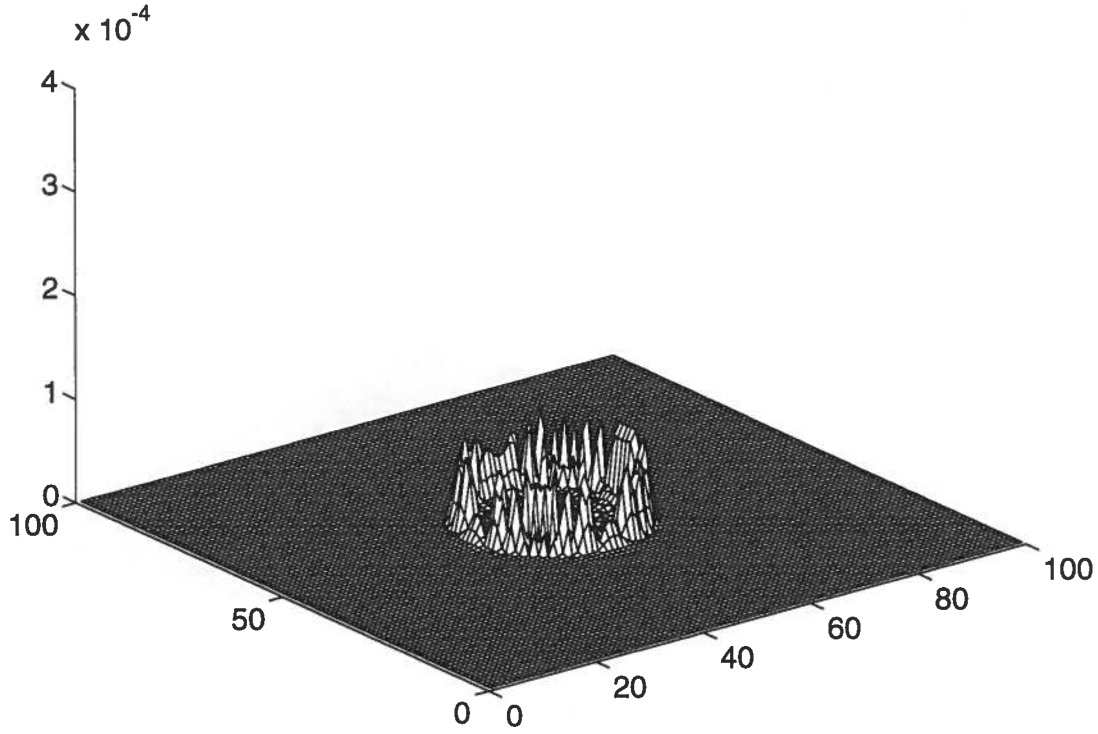

Figure 8.3: Resultat etter 250 iterasjoner.

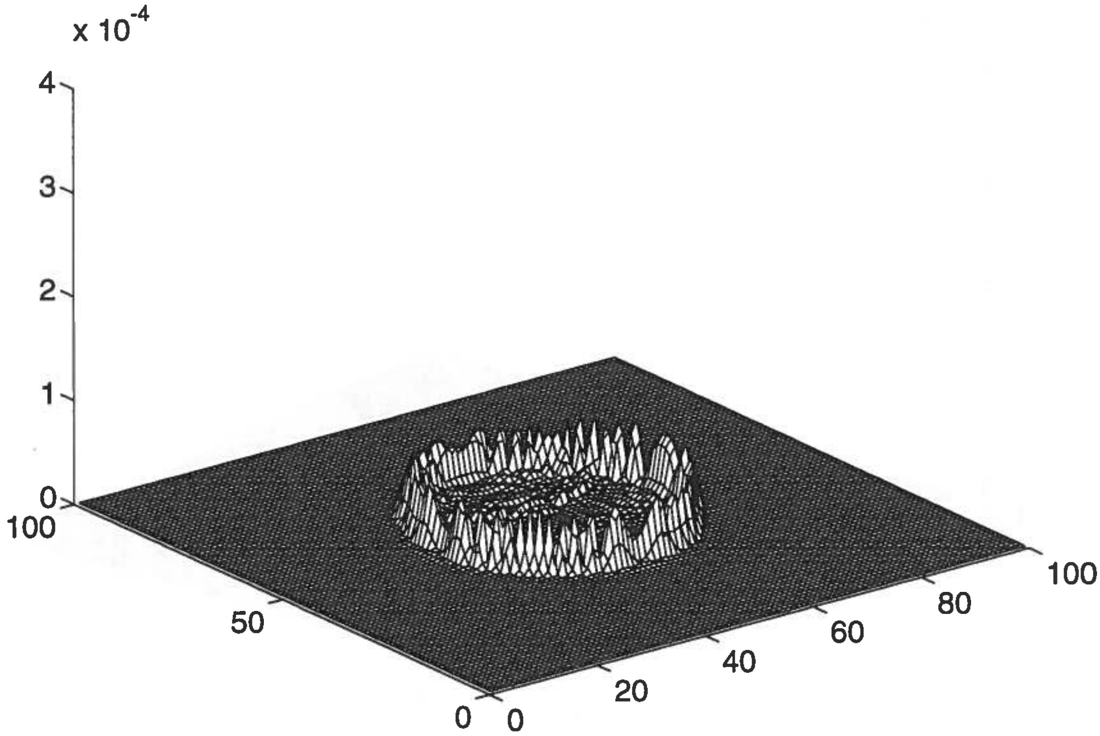

Figure 8.4: Resultat etter 350 iterasjoner.

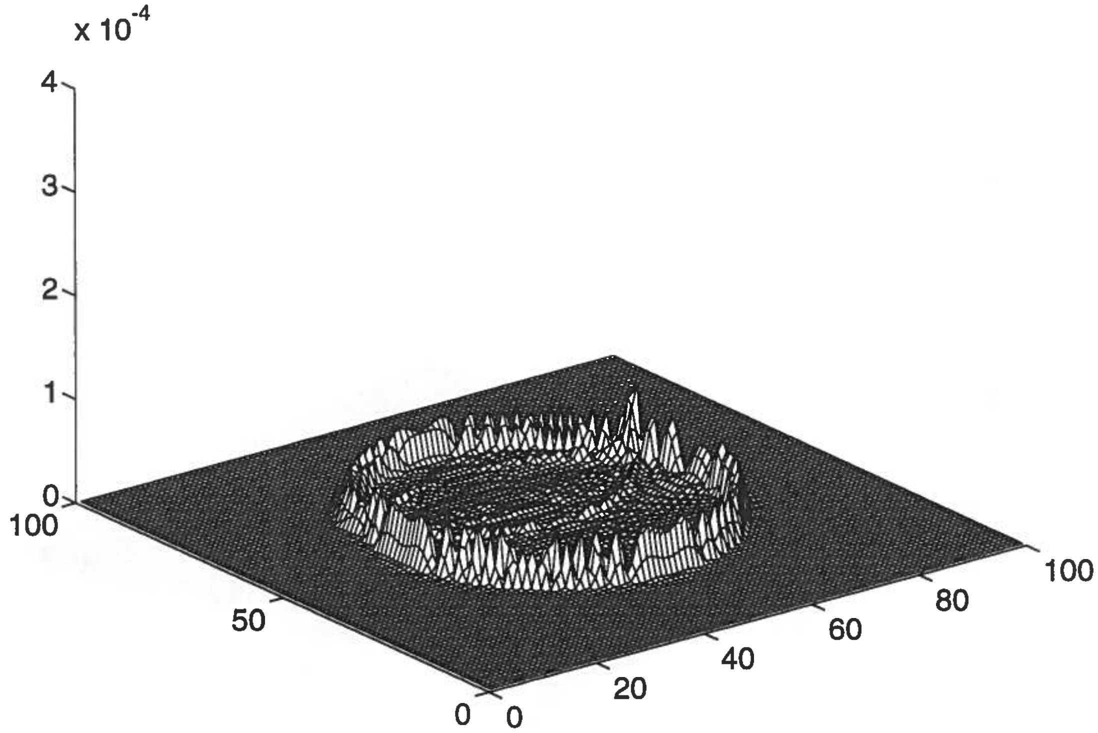

Figure 8.5: Resultat etter 450 iterasjoner.

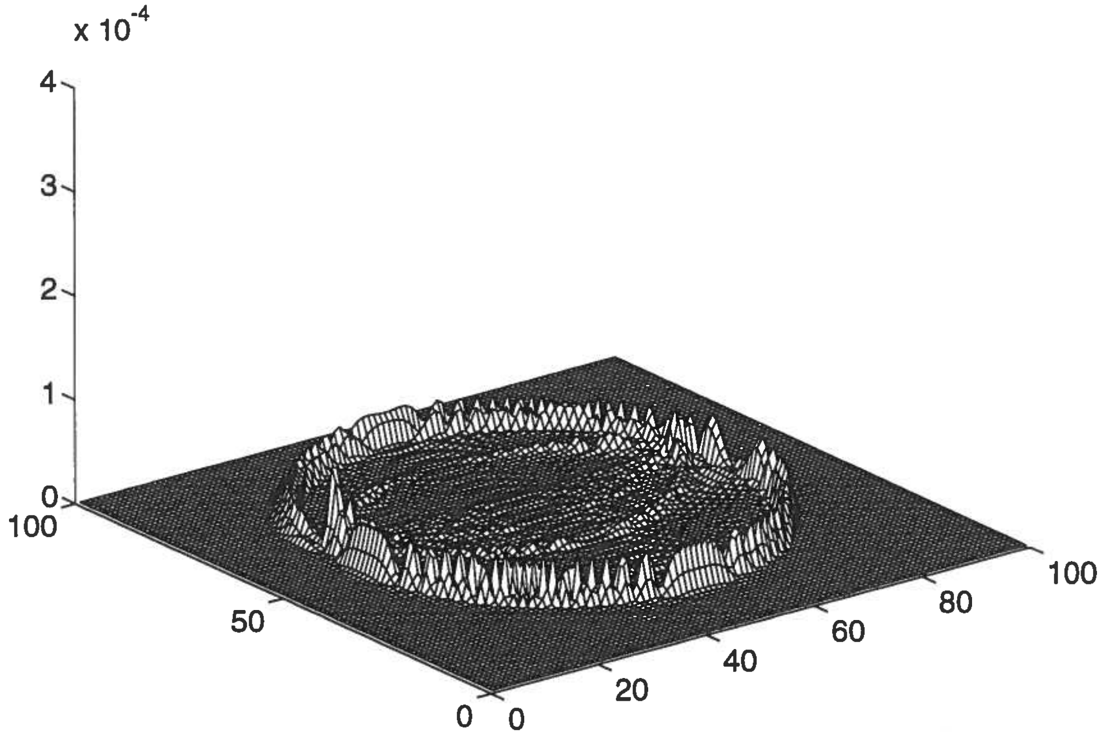

Figure 8.6: Resultat etter 550 iterasjoner.

Chapter 9

Oppsummering og Konklusjon

Tradisjonellt sett er skalarfelt og tensorfelt sett på som uavhengige datatyper, og diskretisering og implementasjon av felttypene er ofte gjort uavhengig av hverandre. Det vil si at man har betraktet de forskjellige felttypene som uavhengige funksjoner fra et diskretisert område til de tilhørende punktverdiene:

```
ringfelt    rf : Domene   -> Ring
modulfelt   mf : Domene   -> Modul
tensorfelt tf : Domene   -> Tensor
```

Der *Domene* er et rom som er diskretisert på en aller annen passende måte. Denne fremgangsmåten har mange ulemper:

- Diskretiseringsmetoder og implementasjonsdetaljer er synlige i alle deler av koden.
- Hvis man ønsker å endre diskretiseringsmetode, må alle tre felttypene implementeres på nytt.
- Derivasjonsalgoritmer må implementeres uavhengig i hvert felt.
- Hvis diskretiseringsmetoden endres må og derivasjonsalgoritmene til alle felttypene endres.

I sum fører alt dette til at hele koden vil bestå av en skog av detaljer som effektivt gjemmer enhver overordnet struktur. En blir dessuten sterkt bundet opp mot en spesiell diskretiseringsmetode, og en endring av denne vil føre til at hele koden må skrives om.

Vi har sett på en alternativ fremgangsmåte som unngår flere av ulempene ved metoden over. Ved hjelp av innkapsling og bruk av parametriserte typer kan vi implementere følgende metode:

```
ringfelt    rf : Domene -> Ring
modulfelt   mf : MODUL  ( rf )
tensorfelt tf : TENSOR ( rf, mf )
```

Det vil si at det kun er ringfeltet som er implementert ved en eksplisitt valgt diskretiseringsmetode. Vi trenger implementere modulfelter og tensorfelter bare en gang, siden vi kan parametrisere dem med den ønskede ringimplementasjon. Vi har sett at hvis vi vet hvordan vi skal derivere en ring R og en modul M over R, kan vi automatisk derivere alle tensorer av typen $T^r_s(M)$. Dette betyr at vi kun trenger implementere en tensorklasse en gang, og siden parametrisere den med en ønsket deriverbar ring R og en ønsket deriverbar modul M. Denne fremgangsmåten har flere fordeler:

- For å endre diskretiseringsmetode i et program, trenger vi kun endre implementasjonen av *rf*. Vi vil da automatisk få oppdatert *mf* og *tf* til å inneholde den nye diskretiseringen. Dette betyr at vi har klart å avgrense den delen av koden som er avhengig av diskretisering til en liten klart avgrenset del av den totale kode.
- Hvis vi ønsker å kjøre et program på en parallell datamaskin, eller vi ønsker å flytte et parallellt program til en ny parallell maskin, trenger vi kun parallellisere eller reimplementere *rf*. Da vil *mf* og *tf* automatisk bli tilpasset den nye maskinarkitekturen. Siden *rf* er kun en liten avgrenset del av et program sin totale kode, vil dette være mye lettere enn å parallellisere både *rf*, *mf* og *tf*.
- Når en har utviklet et bibliotek av forskjellige implementasjoner for *rf*, og man har en implementasjon av *mf* og *tf*, er det lett å utvikle og endre et program som utfører beregninger på tensorfelter.

I denne oppgaven har vi begrenset oss til å implementere et spesialtilfelle av deriverbare tensorer. Dette spesialtilfellet er at vi kun har implementert deriverbare tensorfelt som er definert ut fra deriverbare ringer. Det kan illustreres på følgende måte:

```
ringfelt    rf : Domene -> Ring
modulfelt   mf : MODUL  ( rf )
tensorfelt tf : TENSOR ( rf )
```

Det vil si at vi ikke har parametrisert *tf* med en modul over *rf*. Dette er ikke nødvendig, siden vi med vår begrensning entydig har gitt *mf* ut fra *rf*. Som vi husker skjedde dette når vi definerte basisen til *mf* å være mengden av alle lineært uavhengig, ikke trivielle derivasjonsoperatorer i *rf*. Denne avgrensingen gjør at vi kun kan arbeide med modulfelt som har en felles basis i hele feltet. Innenfor global analyse tilsvarer dette at vi har begrenset oss til å kun arbeide med felter som eksisterer på *parallelliserbare mangfoldigheter*. Dette er ikke en alvorlig begrensning, siden de fleste program som utfører beregninger på tensorfelt kun arbeider på disse mangfoldighetene.

Det er fortsatt mye interessant arbeid som kan gjøres, og denne oppgaven må kun betraktes som en forsiktig start på dette. Det er likevel mye som tyder på at vi har funnet en interessant innfallsvinkel til feltet. Det har etter hvert dukket opp en del rapporter fra andre institusjoner som og ønsker å benytte høynivå programmeringsspråk for å utvikle bedre kode innen numerisk databehandling. Det viser seg ofte at dette ikke er en triviell oppgave, siden det ofte ikke er opplagt hvilke abstraksjoner en skal benytte. Det virker som dette bunner i et generellt metodeproblem. Man har ikke en helhetlig fremgangsmåte/metodikk for å finne gode abstraksjoner, og ofte er det metoden med prøving og feiling som blir benyttet. Innenfor programmeringsteknologi har dette lenge vært et eget forskingsomåde, og det er etterhvert utviklet en god del kunnskaper om abstrakte datatyper det nært beslektede området objektorientering. I denne oppgaven har vi prøvd å kombinere kunnskap fra feltet numerisk databehandling med kunnskap fra feltet programmeringsteknologi, for på denne måten å finne de rette abstraksjonene. Det vi har lært og utviklet i denne oppgaven tyder på at vi er på rett vei.

Bibliography

[1] Abraham R., Marsden J.E. , Ratiu T.. *Manifolds, Tensor Analysis, and Applications.* Springer-Verlag; New York 1988

[2] Boyle James M. , Harmer Terence J.. *Functional Specifications for Mathematical Computations* Argonne National Laboratory

[3] Budge K. G. ,Peery J. S. ,Robinson A. C. . *High-Performance Scientific Computing Using C++.* USENIX C++ Conference Proceedings; Portland, Oregon, 1992

[4] Butkov Eugene. *Mathematical Physics.* Addison- Wesley; New York

[5] Dahl Ole-Johan. *Verifiable Programming.* Prentice Hall; UK 1992

[6] Haveraaen M., Madsen V., Munthe-Kaas H. : *Algebraic Programming Technology for Partial Differential Equations*, Precedings from "Norsk Informatikk Konferanse 1992".

[7] Keffer T.. *Object-Oriented Numerics, Part 1: Vectors, Matrices and all that Stuff.* The C++ Journal; 1, 1991

[8] Kreyszig Erwin. *Advanced Engineering Mathematics.* Wiley & Sons; USA 1988

[9] Lang Serge. *Linear Algebra I.* Addison Wesley; 1965

[10] Lang Serge. *Algebra.* Addison Wesley; 1969

[11] Lippman Stanley B.. *C++ Primer, 2nd ed.* Adison Wesley; 1991

[12] Marsden J.E., Hughes T.J.E.. *Mathematical Foundations of Elasticity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983

[13] Olver P. J.. *Applications of Lie Groups to Differential Equations.* Springer-Verlag; New York 1986

[14] Skånøy Anne-Mari. *Seismic modelling on massively parallell computers.* Hovedfagsoppgave ved Universitetet i Bergen, Institutt for Informatikk, 1993

[15] Stephenson G.. *Partial Differential Equations For Scientists and Engineers.* Longman Inc.; New York 1985

[16] Walker G.. *Why The Choice Must Be C++.* The C++ Journal; 2, 1992

[17] Watt David A.. *Programming Language Syntax and Semantics.* Prentice Hall ; UK 1991

[18] Wilkinson N.M.. *C++ Return Value Optimization.* The C++ Journal; 2, 1992