GoPT: Pakkemal-mekanismen i Go

Utvidelse av Go med en mekanisme for fleksibel gjenbruk og tilpasning av kode

Kristian Aadalen

Oppgave for graden
Master i Informatikk: Programmering og systemarkitektur
60 studiepoeng

Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

UNIVERSITETET I OSLO

Våren 2021

GoPT: Pakkemal-mekanismen i Go

Utvidelse av Go med en mekanisme for fleksibel gjenbruk og tilpasning av kode

Kristian Aadalen

GoPT: Pakkemal-mekanismen i Go

http://www.duo.uio.no/

Trykk: Reprosentralen, Universitetet i Oslo

Sammendrag

Denne oppgaven utforsker hvordan en mekanisme for fleksibel gjenbruk av deklarasjoner, kalt Pakkemal-mekamismen ("Package Templates"), kan utspille seg i programmeringsspråket Go. Pakkemal-mekanismen er opprinnelig definert for språk som bygger på objektorientert programmering, og har som hensikt å tilpasse og utvide samlinger av klasser ved kompilering. Blant annet tilbyr mekanismen funksjonalitet som å endre navn, utvide de deklarerte klassene direkte, sammenfletting av klasser og overskriving av metode-implementasjoner.

Go skiller seg på en rekke områder fra språk som hovedsakelig bygger på objektorientert programmering, som eksempelvis Java. Go har for eksempel ikke klasser slik som Java, og har på enkelte områder en annen tilnærming til typesjekking. Pakkemal-mekanismen har hovedsakelig fram til nå kun vært utforsket i språk som ligner Java, og vi vil dermed håndtere noen nye og nokså uutforskede aspekter når vi skal se på kombinasjonen av Pakkemal-mekanismen og Go.

Under arbeidet med denne masteroppgaven har vi utviklet en preprosessor som støtter Pakkemal-konsepter i Go, og fungerer som en slags prototype. Som en følge av at Go er forskjellig fra typiske objektorienterte språk, så har en del av arbeidet innebåret å tilpasse de ulike Pakkemal-konseptene slik at det passer bedre til språket. Det resulterende språket, altså kombinasjonen av Go og Pakkemal-mekanismen, tilbyr blant annet funksjonalitet som å "dypt" endre navn på deklarasjoner, utvide med nye attributter og funksjonalitet, sammenflette typer, overskrive funksjons-implementasjoner, samt en mekanisme for generiske typeparametere.

Forord

Jeg ønsker å rette en stor takk til mine veiledere, Eyvind W. Axelsen og Stein Krogdahl, for veldig gode råd og innspill under arbeidet med denne masteroppgaven. Mine veiledere har vært veldig fleksible og tillatt meg å forme oppgaven slik jeg selv har ønsket. Denne perioden har i stor grad vært påvirket av Covid-19, men til tross for dette har vi kunnet gjennomføre ukentlige digitale møter, og veiledningen jeg har fått har vært uvurderlig. Kort tid før innleveringsfristen kom den triste nyheten om at veileder Stein Krogdahl hadde gått bort. Jeg er svært takknemlig for at han var del av dette prosjektet, og lyser fred over hans minne.

Innhold

Kapittel 1

Innledning

Når man arbeider med programvareutvikling, vil man ofte forsøke å dele programmet opp i mindre deler, og samtidig forsøke å gjenbruke deler av programmet som man allerede har laget. Grunnen til at man ofte ønsker å gjenbruke deler av program eller kode, er fordi det kan øke produktivitet, redusere kostnad og forbedre kvalitet på det man lager [12]. Dersom man definerer ulike moduler som kan brukes flere ganger til ulike formål, vil man kunne slippe å definere det samme konseptet på nytt, og i så måte unngå å gjøre den samme jobben flere ganger. For å kunne gjøre dette kreves det at programmeringsspråket eller miljøet man bruker har mekanismer som tillater dette.

De aller fleste programmeringsspråk har mekanismer som på en eller annen måte gjør det mulig å modulisere et program eller problem, men måten dette gjøres på og hva mekanismene er best egnet til varierer. For *objektorienterte språk* så baseres i stor grad modularisering på *klasser*, eller eventuelt *pakker* (som er samlinger av klasser) [1]. På denne måten kan man modellere ulike begreper og samtidig abstrahere bort detaljer knyttet til implementasjon. Dette gjør at man potensielt kan slippe å forholde seg til hvordan en klasse eller pakke er bygget opp, men i stedet kun trenger å vite bruksområder og hva klassene tilbyr av funksjonalitet. Ved å gjøre abstraheringer vil det også kanskje være enklere å sette ulike elementer (eller moduler) sammen til en større enhet.

I Java, som er et objektorientert språk, kan modellering av et begrep eller konsept gjøres ved å definere ulike klasser, samt ved å samle disse i en pakke. Dersom man for eksempel skulle modellert et veinett mellom byer, altså en oversikt over ulike byer og veier som binder disse sammen, så kunne man for eksempel definert to klasser `By` og `Vei` i en pakke `VeinettMellomByer`. Antageligvis ville man i pakken definert spesifikk funksjonalitet som hadde vært nyttig for dette formålet, og man kunne brukt pakken i forskjellige tilfeller der man ville modellert veinett.

Dersom man i et annet scenario ville modellert noe som delte de samme strukturelle aspektene som i veinett-scenarioet, kunne man kanskje tenke seg å gjenbruke pakken også til dette. Et eksempel på et slikt scenario kunne for eksempel vært en modellering av flyrutenett, der man ville bundet sammen ulike flyplasser med flyruter. Hvis man skulle brukt den samme pakken, ville antagelig det naturlige vært at flyplasser ble representert ved `By`-klassen, og flyruter ble representert ved `Vei`-klassen. Trolig kunne man ønske å døpe om navnene til disse klassene slik at modelleringen ville vært enda mer presis. I tillegg ville det

kanskje også vært gunstig å kunne utvide klassen med ny funksjonalitet som hadde vært spesifikk for flyruter og flyplasser.

Aspekter som å endre navn på klasser og utvide klasser direkte med ny funksjonalitet for ulike formål, altså uten å definere nye klasser, er det liten støtte for med Javas pakke-mekanisme. Dersom man ville gjort endringer som å endre `By`-klassen sitt navn til "Flyplass", ville man antageligvis enten måtte endret deklarasjonen av klassen eller laget en ny *subklasse*[1] i Java. Eventuelt ville man kanskje laget helt nye klasser, og dermed ikke gjenbrukt koden man allerede hadde skrevet, eller brukt komposisjon slik at man hadde "pakket inn" funksjonaliteten i en ny klasse.

Pakkemal-mekanismen (på engelsk kalt "Package Templates" eller "PT") er en moduliseringsmekanisme utviklet ved Universitetet i Oslo der formålet er å tillate fleksibel gjenbruk av samlinger av klasser i objektorienterte språk [16]. I eksemplene over kunne denne mekanismen vært et nyttig verktøy. For eksempel ville Pakkemal-mekanismen støttet funksjonalitet som å gi klassene nye navn, slik at man ville kunne endret klasse-navn fra "By" til "Flyplass" og "Vei" til "Flyrute". I tillegg ville man kunne utvidet klassene direkte med spesifikk funksjonalitet for "flyrutenett"-modelleringen.

Hovedkonseptet for Pakkemal-mekanismen går ut på at man ønsker å definere struktur og funksjonalitet som senere kan gjenbrukes til ulike formål. I stedet for å først definere en spesifikk pakke for veinett og deretter bruke denne spesifikke pakken for å modellere et flyrutenett, ville det sannsynligvis vært mer hensiktsmessig å finne fellestrekkene fra de to eksemplene og laget en felles mal som inneholdt den underliggende strukturen, samt generell funksjonalitet. Deretter kunne man i de to scenarioene brukt den samme malen som et utgangspunkt og videre gjort tilpasninger og utvidelser som ville vært spesifikke for de to eksemplene. Dette ville potensielt ført til at man hadde sluppet å definere den underliggende strukturen flere ganger, og på den måten fått gjenbrukt den samme koden.

Vi vil senere i oppgaven gå i dypere detalj på hva Pakkemal-mekanismen tilbyr av funksjonalitet, og se på konkrete kode-eksempler der mekanismen brukes.

Go er et programmeringsspråk utviklet av Google, og er designet for å kunne takle utfordringene man typisk møter under systemutvikling der. Det har derfor som mål å være spesielt godt å bruke når man skal arbeide med multikjerneprosessorer, nettverkssystemer og internettprogrammering [19]. Go blir stadig mer brukt og i følge Githubs[2] årlige rapport fra 2019 kom det på 10. plass over språk med størst vekst fra forrige år, med en økning på 147 prosent [29].

Go skiller seg fra Java på flere områder. Blant annet har Go en annen tilnærming til dette med objektorientering, der språket for eksempel ikke har klasser. Java er i stor grad basert på polymorfisme der språket har et nokså tydelig klasse-hierarki. Utviklerne av Go har derimot bevisst designet sitt språk slik at typene ikke baseres på et slikt hierarki som i Java [19]. Mange vil nok kanskje mene at det på dette området ligner mer på programmeringsspråk som C. Likevel skal vi senere i oppgaven se at Go har aspekter som tillater programmering

[1] En klasse som bygger videre på en annen klasse (uttrykt ved bruk av nøkkelordet "extends" i Java) og som dermed "arver" det som er deklarert for klassen den bygger videre på

[2] Github er blant de største lagringsstedene for programkode

på en objekt-basert måte. En annen nokså distinkt forskjell fra Java er aspekter ved *typesjekkingen*, noe som vil påvirke denne oppgaven.

Programmeringsspråket har åpen kildekode og ligger tilgjengelig på GitHub [27], og det er slik at alle kan bidra til å utvikle språket videre. De har også sine "prøveprosjekter" åpent tilgjengelig, noe som kan være nyttig for å få et innblikk i hvordan språket videreutvikles. Dette er noe vi skal se at også har vist seg å være nyttig i dette prosjektet.

De første publikasjonene om Pakkemal-mekanismen tok utgangspunkt i Java, og mekanismen har også vært implementert i språket ved hjelp av verktøyet *JastAdd* [21]. Siden dette har det også vært forsket videre på hvordan mekanismen kan utspille seg i andre programmeringsspråk som Boo [24], C# [14] og Groovy [2]. Fellesnevneren for disse språkene er at de ligner nokså mye på Java, og vi kan nok kategorisere alle som objektorienterte språk.

I denne oppgaven vil vi se på hvordan Pakkemal-mekanismen kan utspille seg i Go. Siden dette språket er nokså ulikt de nevnte objektorienterte språkene, så kommer vi til å håndtere noen nye og hittil nokså uutforskede aspekter. Dette vil både innebære å se på hvordan vi vil bli nødt til å tilpasse Pakkemal-konseptene for å passe programmeringsspråket, og hva dette kanskje kan tilføre språket av nye aspekter. Vi vil også forsøke å redegjøre for om Pakkemal-mekanismen i Go kan resultere i noe som faktisk har en nytteverdi.

Under arbeidet med denne oppgaven valgte vi å lage en prototype. På denne måten kunne vi få et bedre grunnlag for å vurdere hvordan en introduksjon av Pakkemal-mekanismen i Go kan utspille seg. Dette har innebåret å videreutvikle en eksisterende *preprosessor*[3] som vil skrive om programkode som kan inkludere en ny type Go-syntaks til programkode som kun inneholder den gjeldende, offisielle Go-syntaksen. Oppgaven har dermed vært nokså teknisk og praktisk, siden det har betydd at vi har skrevet en god del kode. Samtidig har det innebåret å undersøke arkitektoniske aspekter.

Som nevnt tidligere så er et av hovedmålene for Pakkemal-mekanismen å kunne definere struktur og funksjonalitet som senere kan gjenbrukes til ulike formål. I denne oppgaven vil vi derfor forsøke å introdusere en ny måte for gjenbruk av samlinger av datastrukturer og funksjonalitet i Go, hvor vi i hovedsak tar utgangspunkt i konsepter som allerede eksisterer i språket. Vi vil ha som mål at introduksjonen av dette kan føre til nye måter å utvikle Go-program der vi blant annet kan endre navn på deklarerte datastrukturer, samt utvide disse til spesifikke formål. Ideelt ville vi ønske at dette kunne føre til gjenbruk av kode der vi ellers ville laget noe fra bunnen av, eller at vi ville gjort det på en mer kompleks måte.

Selv om vi i hovedsak tar utgangspunkt i konsepter som allerede er definert i Go, så vil vi også introdusere noen nye. Blant annet vil vi muliggjøre bruk av *generiske typeparametere*, noe som til dags dato ikke er støttet i det offisielle språket.

Under følger en liste over hva vi ønsker å oppnå med oppgaven:

- Undersøke hvordan Pakkemal-mekanismen og konsepter som er definert for den passer et språk som Go. Dette innebærer også å se på hvordan disse konseptene må tilpasses for å være fornuftige i språket.

[3] En preprosessor er et program som endrer på et innsendt program før selve kompilatoren håndterer det [23]

- Utforske hvordan Go sitt typesystem påvirker Pakkemal-mekanismen, både om dette legger begrensninger for hva som er mulig og om dette kan tilføre nye nyttige aspekter.
- Forsøke å implementere en mekanisme i språket som støtter gjenbruk av kode på en fleksibel måte, der vi kan tilby funksjonalitet slik som er mulig med Pakkemal-mekanismen.
- Få en forståelse for om Pakkemal-mekanismen gir en nytteverdi for språk som ikke hovedsakelig bygger på objektorientert programmering, i denne oppgaven representert ved Go.

Oppgaven er strukturert på følgende måte: I kapittel 2 vil vi legge fram hvordan Pakkemal-mekanismen fungerer, og hvordan konseptene for denne mekanismen er definert. Vi vil både vise hva mekanismen bør kunne tilby av funksjonalitet, og et eksempel som viser et typisk scenario hvor vi kan nyttiggjøre oss av mekanismen. I kapittel 3 skal vi se nærmere på programmeringsspråket Go, og beskrive konsepter som er nødvendig bakgrunnsinformasjon for senere kapitler. Kapittel 4 består av en fremlegging av Pakkemal-mekanismen i Go. Her vil det i hovedsak bli vist hva slags funksjonalitet som støttes, og hvordan vi har innlemmet konseptene i programmeringsspråket. I kapittel 5 vil det bli vist hvordan vi konkret implementerte støtten for Pakkemal-mekanismen i Go. Dette inkluderer også å forklare en rekke ulike problemer knyttet til syntaks og semantikk. Kapittel 6 består av evaluering og diskusjon om prosjektet som helhet, samt hvordan mekanismen faktisk utspilte seg i Go. Dette inkluderer også å se på aspekter som antageligvis har et forbedringspotensial. I kapittel 7 konkluderer vi arbeidet, og kommer med forslag til videre arbeid.

Kapittel 2

Pakkemal-mekanismen

Pakkemal-mekanismen ble opprinnelig definert som en alternativ variant for gjenbruk av samlinger av klasser i objektorienterte språk [16]. Som nevnt i innledningen er formålet med mekanismen at vi på et "pakke-nivå" skal kunne ha større fleksibilitet enn det som er tilfellet med for eksempel pakke-mekanismen i Java. I dette kapittelet vil vi gå litt i dybden på hvordan Pakkemal-mekanismen fungerer og hva den tilbyr. Siden de første artiklene om mekanismen tok utgangspunkt i Java, vil vi også bruke dette språket videre i kapittelet.

2.1 Graf-eksempel

Et eksempel som er mye brukt for å vise nytten til Pakkemal-mekanismen, hentet fra [16], omhandler bruk av en graf. I kode-eksempelet nedenfor er det to klasser som er samlet i en *pakkemal* (opprinnelig definert som "template"). De to klassene følger i all hovedsak vanlig Java-syntaks, mens dette "pakkemal-skallet" rundt klassene er ny syntaks definert for Pakkemal-mekanismen. Denne nye syntaksen er slik at vi angir nøkkelordet `template` etterfulgt av navnet på pakkemalen.

```
template Graf{
   class Node{
      Kant førsteKant, sisteKant;
      void settInnKant(Node til) { ... }
   }

   class Kant{
      Node fra, til;
      Kant forrigeKant, nesteKant;
      void fjernMeg() { ... }
   }

}
```

I eksempelet over består `Graf` av en "node-klasse" og en "kant-klasse" med støtte for å legge til og fjerne kanter. Som vi senere skal se kan en slik struktur brukes for å løse reelle problemstillinger, som for eksempel veinett-eksempelet som ble beskrevet i innledningen. I dette eksempelet ville vi for eksempel kunne

brukt node-klassen for å representere byer, og kant-klassen for å representere veier.

For å kunne bruke innholdet i `Graf` er vi nødt til å ha en måte å lage instanser av klassene. Måten vi gjør det er ved å *instansiere* pakkemalen ved å bruke nøkkelordet `inst` etterfulgt av navnet på pakkemalen. Dersom vi også ønsker å endre klassenes navn, vil dette skje i samme setning. Følgende setning vil instansiere `Graf`, og endre klassenavn fra "Node" til "By" og "Kant" til "Vei":

```
inst Graf with Node => By, Kant => Vei
```

Det som skjer ved instansiering av `Graf` er at det ved kompilering blir tatt kopi av alle klassene. Deretter vil vi, på kopiene, gjøre de spesifiserte endringene av navn. Når dette er gjort, vil vi bytte ut setningen med de resulterende klassene, og vi ender dermed opp med to klasser `By` og `Vei`.

Videre skal vi se på et litt mer komplett eksempel der den samme instansieringen finner sted. Her vil vi definere en pakke `VeinettMellomByer`, og i tillegg til endringer av navn, vil vi også utvide de to klassene. Dette gjøres ved at vi bruker nøkkelordet `addto`, etterfulgt av navnet på den resulterende klassen vi skal utvide. Inne i kroppen kan det for eksempel være nye attributter til klassen, samt nye metoder. Siden klassene allerede inneholder deklarasjoner, og vi gjør videre utvidelser, vil de resulterende klassene inneholde unionen av innholdet i "addto-setningen" og det som allerede var deklarert for klassen.

```
package VeinettMellomByer{
   inst Graf with Node => By, Kant => Vei;

   addto By {
      String navn;
      void leggTilNaboby(By b){
         ...
         int n = førsteKant.lengde;
         ...
         Vei v = settInnKant(b);
         ...
      }
   }

   addto Vei {
      int lengde;
      int finnAvstand(){
         ...
         String s = til.navn;
         ...
         int n = nesteKant.lengde;
         ...
      }
   }
}
```

I koden over endres klassen `Node` sitt navn til "By" og `Kant` til "Vei", og i tillegg utvides klassene med hvert sitt attributt og hver sin klasse. Deklarasjonen `n` i den nye metoden `leggTilNaboby` bruker en kombinasjon av attributtet

`førsteKant`, som var definert i den opprinnelige `Node`, og `lengde` som ble definert i "addto"-setningen til `Vei`. Vi kan altså benytte oss av alle attributtene direkte, og vi ser også et tilsvarende eksempel på dette ved deklarasjonen av `s` i metoden `finnAvstand`, der `til` var definert i den opprinnelige `Kant` og `navn` ble definert i "addto"-setningen til `By`. Etter å ha håndtert tilpassingene og utvidelsene av pakkemalen `Graf`, vil den resulterende pakken `VeinettMellomByer` se slik ut:

```
package VeinettMellomByer{
   class By {
      Vei førsteKant, sisteKant;
      String navn;

      void settInnKant(By til) { ... }
      void leggTilNaboby(By b) { ... }
   }

   class Vei {
      By fra, til;
      Vei forrigeKant, nesteKant;
      int lengde;

      void fjernMeg() { ... }
      int finnAvstand() { ... }
   }
}
```

Endringene og utvidelsene som gjøres av `Graf` når `VeinettMellomByer`-pakken defineres kan sees på som nokså spesifikke, og som samtidig er nyttige for det konkrete scenarioet "veinett mellom byer". En metode som det å finne avstand ville antageligvis ikke vært så passende å ha definert direkte i "Graf"-pakkemalen, siden det blant annet ville fordret at alle kanter måtte hatt et attributt om avstand mellom noder. Dette ville vært lite hensiktsmessig dersom vi for eksempel skulle brukt grafen til å modellere et familietre, der nodene ville representert personer og kantene representert personenes relasjon. I et slikt scenario der vi kun er interessert i et familiemønster, ville antageligvis et attributt om avstand vært irrelevant.

I innledningen ble det nevnt et scenario der vi ville modellere et nettverk av flyplasser og flyruter. Det vi nå kunne gjort var på nytt å instansiere pakkemalen `Graf`, der vi hadde gjort de spesifikke endringene og utvidelsene som hadde vært nyttig for dette scenarioet. Vi ville da definert en annen pakke ved å for eksempel skrive `package FlyruteNett`, og lagt til funksjonalitet som å kunne hente ut avgangstider og lignende. Dette hadde gjort at vi hadde sluppet å definere den underliggende strukturen på nytt, og vi ville gjenbrukt den samme koden i de to scenarioene.

2.2 Designmål for Pakkemal-mekanismen

I denne delen vil vi beskrive diverse "designmål" for Pakkemal-mekanismen. Dette er ulike aspekter som setter krav og viser hvordan mekanismen skal fungere med bruk av Java. Målene fra 2.2.1 til 2.2.5 er i stor grad basert på måten det

ble forklart og vist i [16].

2.2.1 Endring av navn på klasser og tilhørende attributter

Ved instansiering av en pakkemal skal det være mulig å kunne endre navn på klasser som er definert inne i denne. Dette ble vist i 2.1 der vi endret begge klassenes navn. I tillegg skal vi kunne endre navnene til klassens attributter og metoder. Den foreslåtte måten å gjøre dette på er ved å liste de opp samtidig som vi endrer selve klasse-navnet. Under følger et eksempel der vi instansierer pakkemalen `Graf` fra 2.1 med å også endre navn på klassenes attributter:

```
inst Graf with
   Node => By(førsteKant -> førsteVei, sisteKant -> sisteVei),
   Kant => Vei(fra -> fraBy, til -> tilBy);
```

Vi kan se her at node-klassens attributter vil hete henholdsvis "førsteVei" og "sisteVei", og kant-klassens attributter vil hete henholdsvis "fraBy" og "tilBy" etter instansiering. I tillegg kunne det kanskje vært naturlig å endre node-klassens metode "settInnKant" til for eksempel "leggTilBy", noe som ville vært gjort på samme måte som endringen av klassenes attributter over.

Et viktig poeng med denne funksjonaliteten er at endringer av navn baseres på semantiske bindinger. Det vil si at vi for eksempel ikke kunne direkte oversatt alle forekomster av et gammelt navn med et nytt navn, men vi måtte i stedet funnet den faktiske deklarasjonen av det vi vil endre navn på og samtidig endre alle referanser til det som ble endret. Mekanismen er med andre ord kraftigere enn kun tekstlige endringer.

2.2.2 Bevaring av klassehierarki

Pakkemal-mekanismen kan inneholde klasser som arver av andre klasser. Med andre ord kan det godt være slik at vi i en pakkemal har en klasse `B` som er en subklasse av en klasse `A`. Etter en instansiering kan vi ha utvidet begge klassene, og sånn sett endret klassene mye fra slik de opprinnelig var definert. Uavhengig av dette skal det være slik at etter instansiering og alle tilpasninger er tatt hånd om, skal fortsatt `B` (som nå godt kan hete noe annet) være en subklasse av `A`. La oss anta dette eksempelet:

```
template T {
   class A {
      ...
   }

   class B extends A {
      ...
   }
}

package P {
   inst T with A => Person, B => Student;
}
```

Etter at vi i `P` har instansiert `T`, skal det være slik at `Student` er en subklasse av `Person`.

2.2.3 Parallell utvidbarhet

I forbindelse med instansiering av en pakkemal, skal vi kunne utvide klasser med nye attributter og metoder. Dette ble vist i veinett-eksempelet i 2.1 der vi utvidet klassene `By` og `Vei`. Etter instansieringen er ferdig skal vi kunne aksessere attributtene direkte uten noen form for *casting*[1]. Det ble også vist i eksempelet, der vi direkte aksesserte attributter som var definert i de opprinnelige klasse-deklarasjonene og attributter som ble definert i "addto"-setningene.

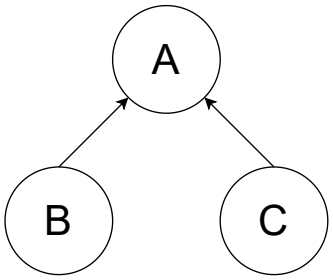
Syntaksen for å utvide klasser er for øvrig endret fra slik det var definert i [16]. Tidligere var det på denne formen:

```
class KlasseNavn adds { ... }
```

Med syntaksen over kan det muligens virke som om vi deklarerer en ny klasse, noe som ikke er tilfellet, siden vi bygger direkte videre på klassen. Derfor er dette endret til syntaksen under, slik at det forhåpentligvis er mer intuitivt at vi utvider en eksisterende klasse:

```
addto KlasseNavn { ... }
```

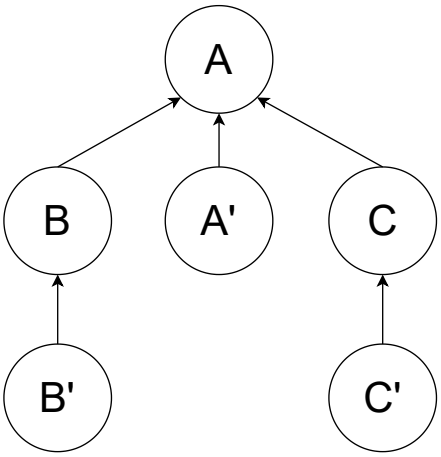
Dette med "parallell utvidbarhet" berører også et problem med å utvide klassehierarkier. La oss anta en klasse `A` og to subklasser `B` og `C` som arver av `A`. Dette gir oss følgende tre:



Figur 2.1: Klassehierarki før utvidelser

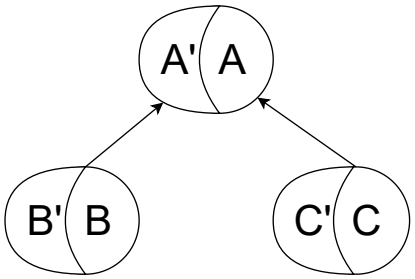
Dersom vi nå, med bruk av Java, ønsker å utvide hver av klassene, så er en mulig løsning å lage nye subklasser av hver av de allerede deklarerte klassene. Dette vil gi oss treet under:

[1] Casting er midlertidig konvertering av en variabels type til en annen type



Figur 2.2: Utvidet klassehierarki med bruk av subklasser i Java

Problemet med å gjøre det på denne måten er at utvidelsene vi har gjort i klassen `A'` ikke reflekteres direkte i `B` og `C` (samt subklasser av disse). Dette gjør for eksempel at vi i `B'` ikke har direkte tilgang på attributter som er deklarert i `A'`. Det er fordi det er ikke er noen relasjon mellom de to klassene, og siden vi i Java kun kan arve fra én klasse, er det heller ingen måte vi kan få en subklasse av `B` eller `C` til å arve fra `A'`. Med bruk av Pakkemal-mekanismen ønsker vi å gjøre det slik at `B` og `C` (samt deres subklasser) får direkte tilgang på utvidelsene i `A'`. Vi vil dermed ende opp med et tre som dette:



Figur 2.3: Utvidet klassehierarki med bruk av Pakkemal-mekanismen

I treet over vil utvidelser for `A`, som er representert som `A'`, reflekteres i `B` og `C`, i tillegg til i `B'` og `C'`. Dermed vil vi kunne aksessere nye deklarasjoner for `A` direkte i dens subklasser. Det er fordi `A` og `A'` er samme node, i motsetning til hva som var tilfellet i treet der vi introduserte nye subklasser.

2.2.4 Flerbruk

Innen samme program, og samme skop, skal vi kunne instansiere en pakkemal flere ganger til ulike formål. De ulike formålene vil som oftest kreve ulike tilpasninger og utvidelser, noe mekanismen må ta høyde for og støtte. For å illustrere dette så kan vi ta utgangspunkt i pakkemalen `Graf` fra 2.1 (for enkelhets skyld uten metoder). La oss anta at vi ønsker å modellere en by som innebærer

blant annet å videre modellere kollektivtransporten og avløpssystemet i byen. Kollektivtransporten uttrykkes ved å lagre holdeplasser og ruter mellom ulike holdeplasser, og for avløpssystemet ønsker vi å lagre avløpspumpestasjoner og tunneler mellom disse. Vi kan se at de to modelleringene deler en lignende underliggende struktur, og ønsker dermed å instansiere den samme pakkemalen to ganger:

```
template Graf{
   class Node{
      ...
   }

   class Kant{
      ...
   }
}

package By {
   inst Graf with Node => Holdeplass, Kant => Rute;
   inst Graf with Node => Avløpspumpestasjon, Kant => Avløpstunell;
   ...
}
```

I eksempelet over kan vi se at den samme pakkemalen instansieres flere ganger, og i et reelt scenario ville vi nok i tillegg utvidet klassene med nyttig funksjonalitet. Det at den samme pakkemalen brukes flere ganger skal altså ikke føre til problemer. Dog er det verdt å nevne at det å endre navnene i dette scenarioet er avgjørende for at programmet blir gyldig. Dersom vi ikke hadde endret noen navn, ville vi endt opp med flere klasser som ble hetende det samme, noe som ville resultert i et ugyldig program.

2.2.5 Klassesammenfletting

Når vi instansierer pakkemaler, skal vi kunne sammenflette to eller flere klasser fra ulike pakkemaler. Innholdet til den resulterende klassen vil dermed være unionen av attributtene til klassene som ble sammenflettet, samt deres funksjonalitet. Dette muliggjør en form for *statisk multippel arv* [5]. Med det mener vi at vi ved instansiering kan slå sammen to (eller flere) klasser, og få med oss egenskapene fra disse inn i en ny klasse. Det er viktig å merke seg at dette ikke utgjør en form for arv, slik vi kjenner det fra Java, i den forstand at relasjonen til de opprinnelige klassene vil beholdes og være synlige ved kjøring. I stedet vil den nye klassen fremstå som en egen klasse der egenskapene fra de sammenflettede klassene er skrevet direkte i den resulterende klassen. Det er altså er her snakk om "arv" av implementasjon, og ikke etablering av et typehierarki. Multippel arv er for øvrig noe Java ikke støtter.

For å illustrere hvordan klassesammenfletting fungerer kan vi anta to pakkemaler som inneholder hver sin klasse, henholdsvis en "skriver" (kan skrive ut dokumenter) og en "skanner" (digitaliserer et fysisk bilde):

```
template T1{
   class Skriver {
      ...
      void skrivUtDokument(String dokument) { ... }
   }
}

template T2 {
   class Skanner {
      ...
      String skannDokument() { ... }
   }
}
```

Vi kan nå anta at vi ønsker å utnytte funksjonaliteten fra begge, og at vi ønsker å samle de i en pakke for å bruke det som en "kopimaskin" (som typisk først vil skanne et dokument, for så å skrive ut kopier av dette). Dette kan vi løse ved å sammenflette de to klassene `Skriver` og `Skanner`:

```
package KopimaskinPakke {
   inst T1 with Skriver => Kopimaskin;
   inst T2 with Skanner => Kopimaskin;
}
```

Den resulterende klassen `Kopimaskin` vil nå inneholde funksjonalitet for å både skanne dokumenter og skrive ut, noe vi kan anta er krav for en kopimaskin.

2.2.6 Instansieringer av pakkemaler i andre pakkemaler

Inne i pakkemaler skal vi også kunne instansiere andre pakkemaler. Dette fører også med seg at vi skal kunne tilpasse pakkemalene slik som ellers, noe som inkluderer blant annet å kunne gi nye navn, samt utvide med nye attributter og funksjonalitet. La oss anta pakkemalen `Graf` fra tidligere eksempler, for å videre definere en ny pakkemal som kan brukes for geografiske nettverk:

```
template Graf{
   class Node{ ... }
   class Kant{ ... }
}

template GeoNettverk {
   inst Graf with Node => Sted, Kant => Forbindelse;
   addto Sted {
      double lengdegrad;
      double breddegrad;
   }
}
```

Ved å nå instansiere `GeoNettverk` vil vi få alt innhold i `Graf`, der klassene nå heter "Sted" og "Kant". I tillegg inneholder `Sted` attributter for lokasjon. Pakkemalen `GeoNettverk` kan naturligvis igjen instansieres slik som alle andre pakkemaler, og vi kan igjen gjøre tilpasninger og utvide slik vi ønsker.

2.2.7 Typeparameterisering

I en pakkemal skal det være mulig å kunne anta eksistens av klasser, der vi også kan sette visse krav, og hvor disse klassene ikke trenger å bli definert før vi instansierer pakkemalen. Klassene innad i pakkemalen kan dermed bruke disse klassene på samme måte som om de allerede eksisterte. I Java finnes det støtte for dette for én enkelt klasse, mens formålet her er at dette skal gjelde for en hel samling av klasser (pakkemal).

I [3] har det blitt foreslått en måte å tillate typeparametere der vi vil kunne få samme fleksibilitet som andre aspekter ved Pakkemal-mekanismen. Disse typene kaller vi i denne oppgaven for *påkrevde typer* (opprinnelig "Required Types"), og vil bli definert inne i pakkemal-kroppen. Dette gjør vi ved å bruke nøkkelordene `required type` etterfulgt av navnet på typen, og innholdet i en påkrevd type avgrenses av krølleparenteser. Inne i kroppen kan vi angi strukturelle skranker ved å definere metode-signaturer som kreves for typen vi *konkretiserer*[2] med. Følgende er et eksempel der den konkrete typen må ha definert metoden "run" for å kunne konkretisere den påkrevde typen:

```
template T {
   required type R {
      void run();
   }
}
```

Vi kan også kreve at typen er en nominell sub-type av et bestemt grensesnitt. Det gjøres ved å legge til nøkkelordet `extends` og navnet på grensesnittet etter navnet til den påkrevde typen. Kombinasjoner av dette og opplisting av metode-signaturer er også mulig. Videre følger et eksempel på dette, der grensesnittet "Runnable" inneholder metoden "run":

```
template T {
   required type R extends Runnable {
      void stop();
   }
}
```

For å kunne konkretisere `R` må den konkrete typen ha definert både `run`- og `stop`-metodene. Siden `R` er en nominell sub-type av `Runnable`, må den konkrete typen i tillegg eksplisitt implementere dette grensesnittet. Denne konkretiseringen skjer for øvrig samtidig som vi instansierer pakkemalen, og videre følger et eksempel på instansiering av pakkemalen der vi kan se at `K` innfrir kravene til den påkrevde typen:

[2] Ved instansiering spesifiserer vi en faktisk konkret type som konkretiserer den påkrevde typen, og som er dermed er typen som blir brukt som den påkrevde typen

```
class K implements Runnable {
   void run() {
      ...
   }

   void stop() {
      ...
   }
}

package P {
   inst T with R <= K
}
```

Dette at påkrevde typer kan tilby fleksibilitet slik som andre Pakkemal-aspekter er noe vi skal se nærmere på senere i oppgaven, men i korte trekk vil det si at vi har mulighet til å for eksempel endre navn og utvide med nye krav. I tillegg vil vi for eksempel kunne sammenflette flere påkrevde typer til én.

2.2.8 Overskriving av metoder

I [4] ble det muliggjort funksjonalitet som tillot å kunne overskrive metoder som allerede var definert i en pakkemal. Dersom vi antar at vi i en pakkemal har en klasse som inneholder en metode `m`, så kan vi med bruk av `addto` bytte ut denne med en ny deklarasjon. Anta følgende pakkemal:

```
template T {
   class K {
      string m() {
         return "Hello"
      }
   }
}
```

Metoden `m` vil i dette tilfellet altså returnere "Hello". Dersom vi i utvidelsen av `K` overskriver metoden, vil vi kunne endre hva som returneres. Når vi skal overskrive en metode, er vi nødt til å eksplisitt markere dette med annotasjonen "@TOverride". I eksempelet under overskriver vi metoden slik at det nå er "World" som returneres:

```
inst T
addto K {
   @TOverride
   string m() {
      return "World"
   }
}
```

Etter instansiering og håndtering av "addto"-setningen, vil kall på `m` nå returnere "World". Dette ville også vært tilfellet dersom vi for eksempel hadde kalt `m` andre steder i pakkemalen `T`.

Det er fortsatt mulig å kalle på den opprinnelige deklarasjonen av `m`, noe som

gjøres ved å legge til "`tsuper.`" foran metode-kallet. I koden under overskrives `m` slik at den returnerer en konkatenert streng ved å kalle på den opprinnelige metoden først, for så å legge til " World" etter:

```
inst T
addto K {
   @TOverride
   string m() {
      return tsuper.m() + " World"
   }
}
```

Dersom vi nå kaller på `m`, vil strengen "Hello World" bli returnert.

Kapittel 3

Programmeringsspråket Go

Google utviklet Go som følge av at de mente det ikke fantes et programmeringsspråk og rammeverk som passet til utfordringene de møtte under utvikling i deres bedrift [30]. De mente at ved å velge et av de etablerte programmeringsspråkene, var de nødt til å velge mellom effektiv kompilering, effektiv kjøring eller enkel programmering, og der man altså ikke kunne få alle disse egenskapene på en gang. Dette ville de løse ved å lage Go, og språket har som følge av dette kjennetegn fra en rekke ulike språk, som isolert sett kanskje ikke ligner så mye på hverandre.

Vi vil i dette kapittelet forsøke å forklare konsepter og aspekter i Go slik at vi senere vil være i stand til å kunne se språket i sammenheng med Pakkemal-mekanismen. Siden Pakkemal-konsepter til nå har vært definert ved bruk av Java, vil vi også gå litt inn på aspekter hvor Java og Go divergerer.

3.1 Typer og konstruksjoner i språket

Noe av motivasjonen for å lage Go var å utvikle et språk som var ryddig og enkelt, og som en følge av det har de forsøkt å definere en syntaks i språket som speiler dette [19]. Et eksempel på det kan være antall nøkkelord i språket som kun er 25, som til sammenligning er under halvparten av Javas 51 (dersom vi teller med "true", "false" og "null") [15]. Personer som har erfaring med C fra tidligere vil nok mene at syntaksen i Go ligner en del på C, der de blant annet deler de samme konseptene som *struct* og *peker*. Likevel vil vi se at det er gjort noen endringer, med det formål om å gjøre det enda mer lesbart [20].

3.1.1 Datastrukturer og datatyper

Go har støtte for de fleste "vanlige" datatyper slik vi er vant med fra andre kjente programmeringsspråk. Dette innebærer blant annet en "boolean"-type som representerer om en variabel er sann eller usann. I tillegg har språket en rekke forskjellige typer for tall, der vi kan skille mellom blant annet heltall, desimaltall og om tallene kan være negative eller kun positive. Det er også mulig å spesifisere hvor mange bits variablene skal oppta, noe som også avgrenser hvor store tallene kan være. Språket har i tillegg *strenger* som gjør det mulig å representere tekst.

I tillegg til å ha primitive typer, har Go også sammensatte typer, der en type er satt sammen av flere primitive eller andre sammensatte typer. Vi vil her se nærmere på en del av disse:

- **Array:** Denne tillater å lagre et spesifisert antall elementer i en indeksert liste. Når vi først har deklarert en array, vil vi ikke kunne endre størrelsen senere. Under følger en deklarasjon av en array:

```
var variabelNavn [størrelsePåArray]VariabelType
```

- **Slice:** Denne datatypen ligner veldig på array, men her trenger vi ikke spesifisere lengde. Denne vil dynamisk håndtere dette, og øker kapasiteten automatisk når den trenger det. Vi kan deklarere en slice slik:

```
var variabelNavn []VariabelType
```

- **Map:** Dette er en datastruktur der vi skiller mellom nøkler og verdier. Nøkler blir lagret unikt i en tabell og linkes til de tilhørende verdiene. Hva slags type nøklene og verdiene er bestemmes ved deklarasjon av "map-et". Typen til verdiene kan være hva som helst, mens det for nøkler er et krav om at typen må være sammenlignbar [11]. Slik kan vi deklarere et map:

```
var variabelNavn map[NøkkelType]VerdiType
```

- **Struct:** Structer i Go er nokså likt slik man kjenner det fra C, og er en måte å definere en samling av variabler. Inne i kroppen definerer vi ulike felter som består av en type og et variabelnavn. Her følger et eksempel på en struct-deklarasjon:

```
type Bil struct {
   farge string
   km int
}
```

  En forskjell fra C er at vi kan definere metoder for structene. Dette er funksjoner som tilhører structen, og som kan kalles på via en struct-instans. Syntaksen til en metode er følgende:

  `func (varNavn StructType) metodeNavn(argumentliste) ReturType`

  Videre følger et eksempel på en metode for structen `Bil`:

```
func (b Bil) hentKmStand() int {
   return b.km
}
```

- **Interface:** I Go har vi grensesnitt ("interface") som består av en samling metode-signaturer, og som kan brukes for å sette et slags minimumskrav over metoder som en type må ha definert. Dersom vi deklarerer en variabel typet som et grensesnitt, må objekter som tilordnes til variabelen ha

definert metodene som er spesifisert for grensesnittet. Inne i grensesnittkroppen lister vi opp de ulike metode-signaturene, og følgende kode er et eksempel på et grensesnitt som ville vært gyldig for Bil-structen i eksempelet over:

```
type BilGrensesnitt interface {
   hentKmStand() int
}
```

3.1.2 Variabeldeklarasjoner

Det er flere måter å deklarere variabler på i Go. Vi kan for eksempel gjøre det helt eksplisitt ved å både spesifisere at det er en variabel og hvilken type variabelen skal ha, noe som for øvrig er et krav dersom vi ikke tilordner en verdi til variabelen i samme setning:

```
var a int = 1
```

Go har også støtte for typeinferens slik at dersom vi tilordner en verdi til variabelen i samme setning, vil Go på egen hånd finne ut hva slags type variabelen skal ha. Under følger et eksempel på dette der det også er to tilordninger i samme setning:

```
var a, b = 1, true
```

I eksempelet over vil variabelen `a` bli typet som `int`, og `b` bli typet som `bool`.

Ved å bruke operatoren ":=" vil vi i tillegg ha vi mulighet til å droppe nøkkelordet `var`, og i stedet kun angi variabelnavnet og verdien:

```
a := 1
```

3.1.3 Kontrollstrukturer

Vi vil nå beskrive noen ulike kontrollstrukturer i Go som alle har blitt brukt flittig i den praktiske delen av oppgaven.

Go har kun én type *løkke*, noe som er "for"-løkken. Grunnen til at Go bare har en type løkke, henger nok sammen med at de ønsker å ha et så enkelt og lite språk som mulig. Til sammenligning har Java i tillegg til "for"-løkken flere andre typer løkker, men Go har heller én type løkke som i stedet er nokså fleksibel, og som tillater å uttrykke det som skal itereres på flere ulike måter. Det er blant annet mulig å uttrykke det på en lignende måte som "for"-løkken i Java, nemlig ved å først initialisere en variabel, så definere betingelsen for at løkken kan kjøre og tilslutt hva man skal gjøre etter hver iterasjon: Under følger et eksempel på dette der det itereres ti ganger, og variabelen `i` øker fra 0 til og med 9:

```
for i := 0; i<10; i++ {
   ...
}
```

Vi kan for eksempel også kun definere et sannhetsuttrykk etter nøkkelordet `for`, noe som på den måten blir lignende "while"-løkken i Java. Ved å bruke nøkkelordet `range` vil vi også enkelt kunne iterere over elementer fra en rekke forskjellige datastrukturer, som for eksempel lister eller "maps".

Språket har også "if"-setninger, samt "switch". Dette fungerer på lik måte som det gjør i andre kjente programmeringsspråk. Et aspekt ved "switch" er noe som ble brukt mye under utviklingen av preprosessoren. Dette dreier seg om at vi har mulighet til å lage en "switch"-setning der vi går inn i de ulike blokkene basert på hva slags type variabelen har.

3.2 Typesystemet

Mange programmeringsspråk som er *statisk typet*[1] (som for eksempel Java eller C++) benytter seg av *nominell typesjekking*. Med nominell typesjekking blir kompatibilitet og likhet mellom typer bestemt basert på eksplisitt navnede deklarasjoner. Dersom vi har to typer `A` og `B` med helt likt innhold, men som ikke har noen eksplisitt navnet relasjon, er det slik at det ikke er mulig å tilordne et objekt av `B` til en variabel typet som `A`. Med *strukturell typesjekking* hadde dette vært lov. Her er det ikke eksplisitt navnede deklarasjoner som avgjør om typer er kompatible, men den faktiske strukturen og definisjonene internt i typene. La oss anta et program, med bruk av Go-syntaks, der vi har to structer `A` og `B` som har helt likt innhold, og hvor vi i "main"-funksjonen forsøker å tilordne et objekt av `B` til variabelen `b` som er typet som `A`:

```
type A struct {
   s string
}

type B struct {
   s string
}

func main() {
   var b A = B{"B"}
}
```

I et språk med ren strukturell typesjekking ville dette programmet vært gyldig. Det er fordi `A` og `B` er strukturelt sett like. Et språk som bruker nominell typesjekking hadde dog ikke godtatt et slikt program, og heller ikke i Go hadde dette vært gyldig. Likevel er det ikke slik at Go kun bruker nominell typesjekking, for faktisk er det en hybrid mellom de to variantene.

Typesystemet i Go bruker stort sett nominell typesjekking, men ved håndtering av grensesnitt sjekkes kompatibilitet og likhet strukturelt [31]. Dette vil si at dersom vi definerer et grensesnitt, og har en struct hvor vi også har definert de samme metodene som i grensesnittet, så kan vi tilordne et objekt av structen direkte til en variabel typet som grensesnittet. Under følger et eksempel der nettopp dette skjer. I eksempelet er det et grensesnitt med en metode for å

[1] Statisk typesjekking går ut på at vi kan verifisere at programmet er typesikkert basert på analyser av programteksten, altså før programmet kjøres [22]

hente personnummer. I tillegg er det en struct som har definert samme metode som grensesnittet.

```
type Statsborger interface {
   hentPersonnummer() string
}

type Nordmann struct {
   navn string
   personnummer string
}

func (n Nordmann) hentPersonnummer() string {
   return n.personnummer
}

func main() {
   var ola Statsborger = Nordmann{navn: "Ola Nordmann", personnummer:
       "11111111111"}
}
```

Vi kan se over at i "main"-funksjonen blir et objekt av `Nordmann` tilordnet til variabelen `ola` som er typet som grensesnittet `Statsborger`. Dette er mulig fordi `Nordmann` har definert metoden `hentPersonnummer`.

Det at Go bruker en slags strukturell typesjekking for grensesnitt fører til noen nye og interessante aspekter når vi skal se på språket i kombinasjon med Pakkemal-mekanismen.

3.3 Objektorientering og komposisjon

Som nevnt tidligere så er ikke Go basert på objektorientert programmering på den måten for eksempel Java er. Likevel har språket typer og metoder som tillater en form for objekt-basert programmering. Språket har som nevnt tidligere structer og støtte for å definere metoder tilhørende structer, noe som kan minne litt om klasser og metoder i Java. Java er i stor grad basert på et typehierarki der alle klasser arver fra den forhåndsdefinerte klassen `Object`. Grunnleggerne av Go tok et bevisst valg om at de ikke ønsket et slik typehierarki i språket, noe som førte til at det ikke finnes noe slikt som "subklasser" [19]. Dette er noe av det som skiller Go mest fra språk som hovedsakelig bygger på objektorientert programmering, og grunnleggerne har altså et ønske om en flatere struktur for relasjoner mellom typer, og også hvor de heller ønsker å fremme bruk av grensesnitt og komposisjon.

Komposisjon i Go fungerer både for grensesnitt og structer. Når vi bruker komposisjon med grensesnitt, vil det egentlig si at vi utvider metode-kravene med andre grensesnitts metoder. Dette gjøres ved å liste opp navnet til grensesnittet vi ønsker å bruke komposisjon med i deklarasjons-kroppen til grensesnittet. I kode-eksempelet under brukes det i `B` komposisjon med `A`, noe som fører til at vi må ha definert både `R` og `S` for å kunne implementere `B`.

```
type A interface {
   R() int
}

type B interface {
   A
   S() string
}
```

For structer kan vi bruke komposisjon både ved å angi grensesnitt og andre structer. Dette gjøres på samme måte som for grensesnitt, der vi inne i deklarasjons-kroppen til structen lister opp typene vi vil bruke komposisjon med. Dersom vi angir et grensesnitt, vil vi kunne kalle direkte på metoder som er definert for dette grensesnittet. Dog fordrer dette selvsagt at vi sender med en struct som innfrir disse kravene når vi oppretter et objekt av structen. La oss anta koden nedenfor:

```
type A interface {
   R() int
}

type S struct {
   A
}

type T struct {}

func (t T) R() int {
   return 1
}

func main() {
   s := S{A: T{}}
   fmt.Println(s.R())
}
```

Structen `S` bruker komposisjon med `A`, og `T` har definert den samme metoden som er krevd av `A`. I "main"-funksjonen oppretter vi et objekt av `S` og samtidig et objekt av `T`, og tilordner `S`-objektet til variabelen `s`. Dermed kan vi på linjen under bruke `s` og kalle direkte på `R`, noe som vil resultere i at tallet "1" blir skrevet ut.

I neste eksempel er `A` nå en struct som har definert `R`. Dette gjør at `S` også her får direkte tilgang på metoden `R`:

```
type A struct {}

func (a A) R() int {
   return 1
}

type S struct {
   A
}
```

```
func main() {
   s := S{A: A{}}
   fmt.Println(s.R())
}
```

Vi kan se i eksempelet over at vi i "main"-funksjonen kan kalle direkte på metoden `R` uten at structen `S` eksplisitt har definert denne metoden.

3.4 Pakker

Pakker i Go har samme formål som *biblioteker* og *moduler* har i andre programmeringsspråk, altså å støtte modularisering, separere kompilering og muliggjøre gjenbruk [10]. For hver Go-fil må vi spesifisere hvilken pakke filen tilhører, noe som gjøres ved å skrive `package navn-på-pakken` øverst i filen. Pakken vi definerer trenger ikke å være unik for hver fil, slik at flere filer kan være del av den samme pakken. Dersom vi ønsker å gjøre en pakke tilgjengelig for bruk i andre pakker, kan vi kjøre kommandoen `go install`. Dette vil kompilere pakken, og lage en linkbar objekt-fil. Alle deklarasjoner i pakken som har en identifikator med stor forbokstav, vil dermed være tilgjengelige for bruk utenfor pakken. Deklarasjoner med liten forbokstav vil dog ikke kunne aksesseres utenfor pakken. Dette er måten vi i Go kan kontrollere tilgang på deklarasjoner i pakker.

For å få tak i andre pakker og deres innhold er vi nødt til å importere dem. Dette gjøres ved at vi bruker nøkkelordet `import` etterfulgt av navnet på pakken (som også kan inkludere *filstien*) vi ønsker å bruke. Dersom vi skal importere flere pakker, kan det dog være ryddig å samle dem i samme import, noe som gjøres ved å bruke parenteser. Under følger et eksempel der vi importerer tre forskjellige pakker, og hvor vi også har med spesifiseringen av pakken selv:

```
package pakke1

import (
   "pakke2"
   "pakke3"
   "pakke4"
)
```

For at en kompilering av et program skal bli vellykket, er vi nødt til å bruke alle pakkene. Dersom dette ikke er tilfellet, vil kompilatoren klage på at vi har importert en pakke som ikke brukes. Skulle vi likevel ønske å importere pakken, for eksempel på grunn av sideeffekter dette fører til, kan vi angi et understrektegn foran pakke-navnet. Pakkers avhengigheter, altså bruk av andre pakker, kan ikke være sykliske. Dette er fordi pakker dermed kan kompileres separat og i parallell, noe som faktisk er en av grunnene til at Go sin kompilering er raskere sammenlignet med mange andre kompilerte språk [10].

Go har en rekke hjelpepakker i sitt standardbibliotek. Som vi senere skal se har dette vært veldig nyttig i dette prosjektet, siden det har ført til at vi har kunnet gjenbruke mye funksjonalitet som ville vært komplisert og tatt lang tid å implementere på egen hånd. Go har også et nokså sterkt "samfunn", og lagt godt til rette for at alle som utvikler Go-kode kan laste denne koden opp,

og enkelt kunne gjenbrukes av andre utviklere i sine prosjekter. Utviklere vil typisk laste opp Go-pakker i et versjonshåndterings-verktøy, og andre utviklere kan laste ned pakken til sitt lokale *arbeidsområde* ved å bruke kommandoen `go get` etterfulgt av url-en der pakken ligger. Dette muliggjør på en enkel måte gjenbruk av en stor mengde pakker, noe vi senere skal se at har vært meget gunstig i denne masteroppgaven.

3.5 Generiske typer

Den offisielle versjonen av Go støtter per dags dato ikke generiske typer eller typeparameterisering. Det har i flere år vært diskusjoner i Go-samfunnet om man burde implementere støtte for det, og hvordan en slik mekanisme i så fall burde fungere. Utviklerne av Go har samtidig som vi har arbeidet med denne oppgaven laget en prototype som støtter generiske typeparametere. Dette har de gjort ved å utvikle en slags preprosessor som tar inn Go-kode som kan inkludere generisk typeparameter-syntaks, og som så skriver dette om til vanlig Go-kode. Denne fremgangsmåten er for øvrig lik måten generiske typeparametere i Java ble introdusert [7]. Senere i oppgaven kommer vi til å se nærmere på denne preprosessoren.

Alle som bruker Go har kunnet teste og komme med tilbakemeldinger og forslag på typeparameteriserings-prototypen, noe som har ført til at syntaksen og semantikken har endret seg en god del i løpet av prosjektet. I januar 2021 anså utviklerne av Go prosjektet som ferdig, og opprettet en "language change proposal" [25], noe som er et formelt krav for at en endring eller utvidelse av Go skal kunne implementeres i språket. Forslaget ble godkjent og utviklerne av Go har allerede startet å implementere det i standardspråket, og håper det kan ta del i en beta-versjon mot slutten av 2021 [8].

Senere i oppgaven vil vi sammenligne denne måten å håndtere generiske typer med måten vi definerte for Pakkemal-mekanismen i Go. Derfor vil vi her beskrive kort hvordan generika-forslaget i Go fungerer. Dette er i all hovedsak basert på måten det presenteres i [26].

Med denne nye mekanismen kan vi angi valgfrie typeparametere til type- og funksjons-deklarasjoner. Vi kan stille krav til hvordan typen skal være ved å definere grensesnitt som objekter av den konkrete typen må kunne tilordnes til. Dersom vi ikke stiller noen krav til typen, kan vi bruke det pre-deklarerte navnet `any`.

For å kunne bruke typeparametere i type- og funksjons-deklarasjoner, må vi definere dette etter navnet på deklarasjonen innkapslet i hakeparenteser. Under følger et eksempel på en funksjons-deklarasjon der vi bruker en generisk typeparameter som kan konkretiseres med en vilkårlig type:

```
func F[T any] (p T) {
   ...
}
```

Siden skranken til den generiske typen i funksjonen `F` er `any`, kan altså den aktuelle parameteren i et kall på funksjonen være av hvilken som helst type.

Videre skal vi se et eksempel på en struct-deklarasjon som bruker en generisk typeparameter. I motsetning til typeparameteren i funksjonen `F`, har vi nå definert en skranke for typen `T` som er uttrykt ved grensesnittet `Hashable`.

Skranken er altså strukturelt definert med dette grensesnittet, og inneholder metode-signaturen `Hash`. Dette betyr at den konkrete typen vi bruker for å konkretisere `T` må ha definert denne metoden:

```
type Hashable interface {
   Hash() int
}

type S[T Hashable] struct {
   h T
}
```

Feltet `h` i structen `S` må altså innfri kravet som er satt av `Hashable`. Siden grensesnitt sjekkes strukturelt i Go, betyr dette at typen som sendes inn for `T` må ha definert funksjonen `Hash`. Dette skal vi se i koden under der vi konkretiserer `T` med structen `Person`. Denne structen har definert metoden `Hash`, og innfrir dermed kravet som er definert av `Hashable`. Vi kan videre bruke structen `Person` direkte når vi oppretter et objekt av `S` uten noen form for nominell spesifisering:

```
type Person struct {
   navn string
   adresse string
}

func (p Person) Hash() int {
   var hashValue int
   ...
   return hashValue
}

v := S[Person]{h: Person{"Navn", "Adresse"}}
```

Kapittel 4

Pakkemal-mekanismen i Go

Fram til nå har vi sett på hvordan Pakkemal-mekanismen fungerer i et språk som hovedsakelig bygger på objektorientert programmering. Vi har også sett på sentrale konsepter i programmeringsspråket Go, og aspekter som vil være nødvendig bakgrunnskunnskap i denne oppgaven. I dette kapittelet vil vi se nærmere på hvordan mekanismen kan utspille seg i språket. Dette innebærer i stor grad å forsøke å innlemme konseptene som ble vist i 2.2.

Da vi introduserte Pakkemal-konsepter i Go, tilpasset vi hovedsakelig disse konseptene til å passe Go, i stedet for å endre Go så språket ville passe konseptene slik de opprinnelig ble definert. Dette kommer av at Pakkemal-mekanismen i utgangspunktet er en mekanisme utviklet for objektorienterte språk. Derfor vil det være naturlig at aspekter ved mekanismen ikke nødvendigvis kan implementeres direkte i Go, men i stedet må tilpasses og endres.

Kombinasjonen av Pakkemal-mekanismen og Go resulterer i et nytt språk som vi videre kaller for *GoPT*. Dette er vanlig Go-kode, men som også inkluderer ny syntaks som støtter Pakkemal-konsepter. Med GoPT vil vi blant annet kunne samle deklarasjoner og datastrukturer i en pakkemal. Pakkemaler kan så instansieres og videre bli tilpasset og utvidet for å bli brukt i spesifikke scenarioer. For eksempel kan vi endre navn på typer og attributter, samt overskrive implementasjoner av funksjoner. Vi vil også ha mulighet til å utvide structer og grensesnitt med nye attributter og metoder, og vi kan også nyttiggjøre oss av generiske typeparametere.

I dette kapittelet vil vi hovedsakelig beskrive og diskutere syntaks og semantikk i språket GoPT. Vi utelater blant annet en del tekniske aspekter, samt løsninger på semantiske problemer som i stedet vil bli forklart i senere kapitler.

4.1 Deklarasjon av pakkemaler

Måten vi deklarerer pakkemaler i GoPT er ved å angi nøkkelordet `pttemplate` etterfulgt av navnet på pakkemalen. Inne i en pakkemal kan vi definere ulike typer og funksjoner, i tillegg til andre Pakkemal-konsepter som vil bli forklart senere. Under følger et eksempel der pakkemalen `T` inneholder et grensesnitt, en struct og en metode:

```
pttemplate T {
   type I interface {
      F() string
   }

   type S struct {
      name string
   }

   func (s S) F() string {
      return s.name
   }
}
```

Grunnen til at vi bruker "pttemplate", og ikke for eksempel "template" slik det opprinnelig ble definert, er fordi dette ville ført til store sideeffekter dersom vi skulle implementert det i selve språket. I kildekoden til Go brukes ordet "template" som en rekke identifikatorer, slik at det å nå definere dette ordet som et reservert ord ville ført til å måtte endre store deler av kodebasen. Ved å bruke "pttemplate" kunne vi være sikrere på at vi ikke ville ødelegge annen kode.

4.2 Instansiering av pakkemaler

For å kunne bruke innholdet til pakkemaler er vi først nødt til å instansiere dem. Dette gjøres ved å bruke nøkkelordet `ptinst` etterfulgt av navnet på pakkemalen vi vil instansiere. En instansiering av pakkemalen `T` fra 4.1 kunne sett slik ut:

```
ptinst T
```

Setningen i eksempelet over ville ført til at det under kompilering ville blitt tatt en kopi av innholdet til `T`, og selve setningen ville blitt byttet ut med dette kopierte innholdet. Instansieringer av pakkemaler kan for øvrig gjøres i to forskjellige typer skop. Dette er enten i det vil kaller det *ytre skopet* eller inne i en pakkemal. Det ytre skopet vil i en fil være den ytterste blokken, altså i det samme skopet som vi definerer hvilken pakke filen tilhører. Under følger et eksempel på en Go-fil, der vi ser eksempler på instansieringer i begge de to variantene av skop:

```
package main

pttemplate T1 {
   type S struct {
      s string
   }
}

pttemplate T2 {
   ptinst T1
}
```

```
ptinst T2

func main() {
   ...
}
```

I eksempelet over vil `T2` etter instansiering inneholde det samme som `T1`. Siden vi i det ytre skopet også instansierer `T2`, vil structen `S` etter kompilering være definert i det ytre skopet. Den resulterende koden vil dermed se slik ut:

```
package main

type S struct {
   s string
}

func main() {
   ...
}
```

Valget om å bruke "ptinst", og ikke "inst", har samme begrunnelse som deklarasjon av pakkemaler, altså fordi det ville ført til store sideeffekter.

4.3 Endring av navn ved instansiering

I GoPT vil det også være mulig å endre navn på deklarasjoner inne i en pakkemal. Dette spesifiseres i samme setning som instansiering av pakkemalen, og fungerer slik at vi spesifiserer de nye navnene i en komma-separert liste. Mellom navnet på pakkemalen og den komma-separerte listen bruker vi nøkkelordet `with`, som vi har introdusert i denne oppgaven. Siden en pakkemal kan inneholde flere forskjellige typer, vil også syntaksen for endring av navn variere. Under følger en liste over de ulike måtene å gjøre det på:

- **Endring av navn på struct og dens felter (tomt innhold i parentesene dersom vi ikke vil endre navn på felter):** StructNavn => NyttStructNavn(feltnavn1 -> nyttFeltnavn1, feltnavn2 -> nyttFeltnavn2)
- **Endring av funksjonsnavn:** Funksjonsnavn => NyttFunksjonsnavn
- **Endring av metodenavn:** StructNavn.MetodeNavn => NyttMetodeNavn
- **Endring av grensesnitt (tomt innhold i parentesene dersom vi ikke vil endre navn på metoder):** Grensesnittnavn => NyttGrensesnittnavn(metodenavn1 -> nyttMetodenavn1, metodenavn2 -> nyttMetodenavn2)

Da vi skulle definere syntaksen for endringer av navn, tok vi et bevisst designvalg om at det skulle følge Go sin blokkstruktur. Derfor spesifiserer vi for eksempel endringer av structers felter og grensesnitts metode-signaturer inne i parenteser samtidig som vi endrer selve struct-navnet eller grensesnitt-navnet. Metoder blir i Go deklarert i samme skop som selve struct-deklarasjonen, noe

som er grunnen til at endring av navn på disse ikke spesifiseres sammen med structers felter.

Naturligvis vil vi unngå tvetydige situasjoner ved endringer av navn, altså i situasjoner der vi ikke kan vite hvilken deklarasjon som faktisk skal bli gitt nytt navn. Et eksempel på dette kan være et tilfelle der vi har deklarert både en funksjon og en metode med samme navn. Uten å skille mellom metoder og funksjoner ville det ikke vært mulig å vite hvilken av dem vi faktisk skulle gitt et nytt navn. Derfor har vi valgt å skille mellom syntaksen for endring av navn på metoder og funksjoner i GoPT, der vi for metoder er nødt til å spesifisere hvilken struct den tilhører, mens det for funksjoner kun er nødvendig å spesifisere funksjons-navnet.

For å vise et kode-eksempel hvor det forekommer endringer av navn, kan vi forsøke å gjøre tilsvarende som i 2.2.1, der vi instansierte pakkemalen `Graf` og endret navn på klasser og attributter. Vi vil her også endre navn på metoden `SettInnKant`:

```
pttemplate Graf {
   type Node struct
      førsteKant, sisteKant Kant
   }

   func (n Node) SettInnKant(Node til) { ... }

   type Kant struct {
      fra, til Node
      forrigeKant, nesteKant Kant
   }
   ...
}

ptinst Graf with
   Node => By(førsteKant -> førsteVei, sisteKant -> sisteVei),
   Kant => Vei(fra -> fraBy, til -> tilBy),
   Node.SettInnKant => LeggTilVei
```

4.4 Utvidelse av structer og grensesnitt

Når vi instansierer en pakkemal, er det mulig å utvide structer med nye metoder og felter. For å kunne gjøre dette brukes nøkkelordene `addto struct` etterfulgt av navnet på structen vi skal utvide. Under følger et eksempel på dette:

```
pttemplate T {
   type Person struct
      navn string
   }
}

ptinst T
addto struct Person {
   alder int
```

```
   func (p Person) hentAlder() int {
      return p.alder
   }
}
```

Vi kan se i eksempelet over at structen `Person` utvides med et nytt felt og en ny metode. Structen vil dermed se slik ut:

```
type Person struct {
   navn string
   alder int
}

func (p Person) hentAlder() int {
   return p.alder
}
```

Det er også mulig å utvide grensesnitt med nytt innhold, noe som både kan være nye metode-signaturer og andre grensesnitt-navn. Dette gjøres på omtrent samme måte som for structer, men i "addto"-setningen angir vi nøkkelordet `interface` i stedet for `struct`. Under følger et eksempel hvor det både legges til en ny metode-signatur og et annet grensesnitt-navn:

```
pttemplate T {
   type Student interface
      hentFakultet() string
   }
}

type Person interface {
   hentAlder() int
}

ptinst T
addto interface Student {
   Person
   leggTilEmner(emner []string)
}
```

Etter instansiering av `T` og alle utvidelser er tatt hånd om, må et objekt som skal types som `Student` ha definert `hentFakultet`-, `hentAlder`- og `leggTilEmner`-metodene. Grensesnittet vil dermed se slik ut:

```
type Student interface {
   Person
   hentFakultet() string
   leggTilEmner(emner []string)
}
```

Dette med å kunne utvide grensesnitt er noe som har vært diskutert i artikler, men aldri før vært støttet i andre implementasjoner av Pakkemal-mekanismen. I og med at Go har en typesjekking av grensesnitt som er ganske ulik disse andre språkene, ser vi dog at dette med å kunne utvide grensesnitt kan gi en

nytteverdi i Go. Selv om det kan være nyttig vil det også potensielt kunne føre til utilsiktede sideeffekter, noe vi skal se nærmere på i 5.6.1.

4.5 Sammenfletting av structer og grensesnitt

I 2.2.5 ble det vist Pakkemal-mekanismens egenskap som tillater å sammenflette klasser. Siden Go ikke har klasser i språket, er det naturlig at vi må justere dette litt. Derfor har vi bestemt at vi i GoPT vil kunne sammenflette flere structer til én, samt flere grensesnitt til ett. Måten vi sammenfletter structer eller grensesnitt er ved å døpe om navnene deres til det samme. La oss derfor anta to structer `A` og `B` som befinner seg i hver sin pakkemal. I instansieringene døper vi om structene til å hete "C":

```
pttemplate T1 {
   type A struct {
      a string
   }

   func (a A) HentA() string {
      return a.a
   }
}

pttemplate T2 {
   type B struct {
      b int
   }

   func (b B) HentB() int {
      return b.b
   }
}

ptinst T1 with A => C()
ptinst T2 with B => C()
```

Programmet over ville altså resultert i én struct `C`, seende slik ut:

```
type C struct {
   a string
   b int
}

func (a C) HentA() string {
   return a.a
}

func (b C) HentB() int {
   return b.b
}
```

I det eksempelet over kan vi nå se at structen `C` har begge attributtene fra `A`

og `B`. I tillegg er metodene `HentA` og `HentB` tilknyttet `C` direkte. Med andre ord er nå alle egenskapene til `A` og `B` direkte definert for `C`.

Dersom de to structene hadde hatt identiske felter eller metode-signaturer, ville det resulterende programmet endt opp ugyldig. Dog kunne vi løst det ved å endre navn, slik at vi ikke ville endt opp med duplikater.

Sammenfletting av grensesnitt gjøres på akkurat samme måte som for structer. Følgende kode-eksempel resulterer i ett grensesnitt bestående av metodene `A` og `B`:

```
pttemplate T1 {
   type A interface {
      A() string
   }
}

pttemplate T2 {
   type B interface {
      B() int
   }
}

ptinst T1 with A => C()
ptinst T2 with B => C()
```

Det er for øvrig slik at vi er nødt til å eksplisitt endre til samme navn for at vi skal kunne sammenflette typer. Dette er for å unngå situasjoner der vi utilsiktet sammenfletter typer uten å være klar over det, noe som igjen kan føre til store sideeffekter. Dermed vil følgende program ikke føre til en sammenfletting, og vi vil i stedet ende opp med en feilmelding om at grensesnittet `A` blir "redeklarert":

```
pttemplate T1 {
   type A interface {
      A() string
   }
}

pttemplate T2 {
   type B interface {
      B() int
   }
}

ptinst T1
ptinst T2 with B => A()
```

Det å kunne sammenflette flere typer til én kan gi opphav til en del problemer. En del av disse ligner også på problemer vi møter ved multippel arv, slik som tvetydighet ved for eksempel flere deklarasjoner av samme navn. Dog vil problemene vi typisk møter ved multippel arv være enklere å unngå og håndtere på grunn av måten Pakkemal-mekanismen er definert. Dette er noe vi kommer tilbake til i 5.7.

4.6 Typeparameterisering

I GoPT støttes det en form for typeparameterisering kalt påkrevde typer, som vist i 2.2.7. Dette gjør at vi innad i en pakkemal kan anta eksistens av en type uten at den er konkretisert. I denne oppgaven har vi gjort den antagelsen om at den konkrete typen må være en struct, men vi skal senere diskutere om vi også kunne tillatt andre varianter av typer.

Syntaksen for påkrevde typer er nokså enkel. Inne i pakkemal-kroppen definerer vi påkrevde typer ved å bruke nøkkelordene `required type` etterfulgt av navnet på typen. Så vil vi liste opp krav inne i kroppen, som følger samme syntaks som i kroppen på et grensesnitt. Dette innebærer at vi både kan angi metode-signaturer og grensesnitt-navn. En påkrevd type som har tomt innhold betyr at den kan konkretiseres med hvilken som helst struct.

Vi konkretiserer en påkrevd type ved å angi navnet på den påkrevde typen, og deretter bruke symbolene "<=" etterfulgt av navnet på den konkrete typen. Under følger et eksempel på en påkrevd type som blir konkretisert med en struct som innfrir kravene:

```
pttemplate T {
   required type R {
      String() string
   }

   func UseString(r R) string {
      r.String()
   }
}

type S struct { }

func (s S) String() string {
   return "Hello"
}

ptinst T1 with R <= S
```

I programmet over ville altså den påkrevde typen `R` blitt konkretisert med `S`. Dette ville resultert i følgende program etter instansiering:

```
func UseString(r S) string {
   r.String()
}

type S struct { }

func (s S) String() string {
   return "Hello"
}
```

I programmet over ser vi at parameteren `r` i funksjonen `UseString`, som før var av typen `R`, etter instansiering nå er den faktiske typen `S`.

Da vi presenterte påkrevde typer i 2.2.7, nevnte vi at mekanismen støt-

tet fleksibilitet slik som andre Pakkemal-konsepter. Dette er noe mekanismen i GoPT også støtter. Blant disse er muligheten for å endre navn på selve typen eller på metode-signaturene. Dette gjøres samtidig som vi instansierer pakkemalen, og følger omtrent samme syntaks som endring av navn på structer eller grensesnitt, nemlig slik:

```
PåkrevdTypeNavn => NyttPåkrevdTypeNavn(MetodeNavn -> NyttMetodeNavn)
```

Under følger et eksempel der vi bruker påkrevde typer, og nyttiggjør oss av muligheten for endring av navn:

```
pttemplate PersonPakkemal {
   required type Person {
      HentID() string
   }
}

pttemplate StatsborgerPakkemal {
   ptinst PersonPakkemal with
      Person => Statsborger(HentID -> HentPersonnummer)
}

type Nordmann struct { }

func (n Nordmann) HentPersonnummer() string { ... }

ptinst StatsborgerPakkemal with Statsborger <= Nordmann
```

I `PersonPakkemal` i eksempelet over er det definert en påkrevd type `Person` som krever at metoden `HentID` er definert, og i `StatsborgerPakkemal` instansieres denne pakkemalen. Inne i `StatsborgerPakkemal` gjør vi om navnet på både den påkrevde typen `Person` og metoden `HentID`. Siden vi ikke konkretiserer den påkrevde typen her, så "videreføres" denne. Dermed vil `StatsborgerPakkemal` inneholde en påkrevd type `Statsborger`, som krever metoden `HentPersonnummer`. I det ytre skopet er `Nordmann` definert, og vi kan se at den innfrir alle kravene til den påkrevde typen `Statsborger`. Structen kunne dog ikke konkretisert den påkrevde typen i `PersonPakkemal` uten videre endringer av navn. Dette er fordi den ikke har en metode `HentID`.

Det er også mulig å utvide en påkrevd type med enda flere krav. For dette bruker vi "addto"-setningen, men i stedet for nøkkelordene `interface` eller `struct` bruker vi her `required type` for å spesifisere at det er en påkrevd type vi skal utvide. Innholdet i "addto"-setningen vil være enten grensesnitt-navn eller metode-signaturer. Under følger et eksempel som viser bruk av dette:

```
pttemplate PersonPakkemal {
   required type Person {
      HentID() string
   }
}

pttemplate StudentPakkemal {
   ptinst PersonPakkemal
```

```
   addto required type Person {
      HentFakultet() string
   }
}

type Student struct { }

func (s Student) HentID() string { ... }

func (s Student) HentFakultet() string { ... }

ptinst StudentPakkemal with Person <= Student
```

I eksempelet over kan vi se at `Person` konkretiseres med `Student`. Den påkrevde typen i `StudentPakkemal` krever altså to metoder, noe structen `Student` har definert.

Et siste Pakkemal-konsept som er mulig med påkrevde typer går på sammenfletting av flere typer til én. Dette følger helt lik syntaks som sammenfletting av structer og grensesnitt. Under følger et eksempel på dette:

```
pttemplate T1 {
   required type S {
      F() string
   }
}

pttemplate T2 {
   required type R {
      D() int
   }
}

pttemplate T3 {
   ptinst T1 with S => P()
   ptinst T2 with R => P()
}
```

I eksempelet over vil pakkemalen `T3` ende opp med å inneholde en påkrevd type `P` som vil kreve metodene `S` og `R`. Dersom vi sammenfletter påkrevde typer som inneholder noen like metode-signaturer, så byr ikke dette på noen problemer. Duplikater av metode-signaturer behandles kun som én signatur.

4.7 Overskriving av metoder og funksjoner

I "addto"-setninger har vi mulighet til å bytte ut metoder og funksjoner som er deklarert i en pakkemal med nye implementasjoner. For å tillate dette i Java-versjonen av Pakkemal-mekanismen, ble det brukt en annotasjon "@TOverride" [4]. Det å bruke annotasjoner i Java er noe som er nokså utbredt, slik at det å innføre denne nye annotasjonen der passet godt inn i språket. I Go derimot benyttes ikke annotasjoner på samme måte, og det å bruke alfakrøll i kildekoden byr på store problemer. Likevel er det veldig gunstig å eksplisitt måtte markere

at vi skal overskrive en metode eller funksjon, siden det bidrar til å unngå utilsiktede overskrivinger, noe vi også kommer tilbake til senere. I tillegg gjør det implementasjonen av dette enklere, siden vi på den måten tidlig i en kompilering kan sjekke at en overskriving er gyldig. Derfor bestemte vi at overskriving av metoder og funksjoner i GoPT må eksplisitt markeres med "toverride" før nøkkelordet `func`.

For at en overskriving skal være gyldig, må det være slik at signaturen er identisk med en funksjon eller metode i en pakkemal vi har instansiert i programmet. I kode-eksempelet under vises det gyldig kode der vi overskriver en metode:

```
pttemplate T {
   type S struct { }

   func (s S) String() string {
      return "Opprinnelig metode"
   }
}

ptinst T
addto S {
   toverride func (s S) String() string {
      return "Ny metode"
   }
}

func main() {
   s := S{}
   fmt.Println(s.String())
}
```

Overskrivingen av metoden `String` i kode-eksempelet over fører til at det som blir skrevet ut i "main"-funksjonen er "Ny metode".

Selv om vi tilsynelatende erstatter metoden over, er det faktisk slik at den opprinnelige metoden ikke blir fjernet. Dette er fordi vi fortsatt kan kalle på den opprinnelige, noe som gjøres ved å skrive "tsuper" og punktum foran metodekallet. I neste kode-eksempel overskriver vi metoden `String` slik at den nå også kaller på den opprinnelige metoden:

```
pttemplate T {
   type S struct { }

   func (s S) String() string {
      return "Opprinnelig metode"
   }
}

ptinst T
addto S {
   toverride func (s S) String() string {
      return tsuper.String() + " og ny metode"
   }
}
```

```
func main() {
   s := S{}
   fmt.Println(s.String())
}
```

Metoden `String` vil nå returnere den konkatenerte strengen "Opprinnelig metode og ny metode".

I begge eksemplene over har vi overskrevet en metode tilhørende en struct, og derfor angitt structen `S` i "addto"-setningen. Dersom vi skal overskrive en funksjon, vil det derimot ikke være en struct å referere til, siden funksjoner i Go ikke er tilknyttet noen struct. Måten vi har valgt å løse dette på er at vi i slike tilfeller kan bruke tegnet "understrek" etter `addto`-nøkkelordet. Understrek-tegnet har egentlig ikke noen praktisk hensikt, og i prinsippet kunne vi for eksempel kun hatt nøkkelordet `addto`, og så den tilhørende kroppen. Dette er altså kun et designvalg vi har gjort for denne prototypen.

For at en funksjons-overskriving skal være gyldig, må det finnes en funksjon i en instansiert pakkemal som har lik signatur. I likhet med metoder kan vi også her kalle på den opprinnelige funksjonen ved å bruke nøkkelordet `tsuper`. Under følger et eksempel som i "main"-funksjonen vil skrive ut tilsvarende som forrige eksempel:

```
pttemplate T {
   func String() string {
      return "Opprinnelig metode"
   }
}

ptinst T
addto _ {
   toverride func String() string {
      return tsuper.String() + " og ny metode"
   }
}

func main() {
   fmt.Println(String())
}
```

Kapittel 5

Implementasjon

I dette kapittelet skal vi se nærmere på hvordan vi implementerte støtte for Pakkemal-mekanismen i Go. Dette innebærer å beskrive og diskutere tekniske aspekter ved dette, og konkret hvordan det ble gjort. I tillegg kommer vi til å forklare løsninger på semantiske utfordringer som vi til nå har unnlatt. Selve kildekoden er å finne på [33].

5.1 Hvordan lage prototypen?

I starten av arbeidet med denne oppgaven ble det brukt en del tid på å vurdere og utforske hvordan vi kunne utvikle en GoPT-prototype. Det var naturligvis flere måter å løse dette på, og vi vil videre forklare hvordan denne prosessen var og begrunnelser for hvorfor vi gjorde som vi gjorde. I de følgende avsnittene skal vi se på forskjellige alternativer, og kort diskutere disse, før vi går løs på å forklare selve implementasjonen.

5.1.1 Standardkompilatoren

Noe av det første arbeidet i denne oppgaven gikk ut på å utforske Go sin kildekode, og forsøke å forstå hvordan kompilatoren til språket fungerte. Med nokså snever bakgrunnskunnskap om språket, og det faktum at det var ganske begrensede ressurser på krevende aspekter ved kompilatoren, så var dette en møysommelig prosess.

Vi forsøkte å legge til små utvidelser i språket, og fikk etter hvert til dels til dette. Disse utvidelsene kunne dog ikke sammenlignes med kompleksiteten av Pakkemal-konsepter. Selv om det å implementere Pakkemal-konsepter direkte i hovedspråket kanskje ville vært det mest ideelle, fant vi ganske fort ut at det antageligvis ikke var hensiktsmessig for denne oppgaven. I og med at kjernen av oppgaven går på hvordan kombinasjonen av Pakkemal-mekanismen og Go fungerer, altså ikke *kun* en implementasjon av mekanismen, så vurderte vi det slik at det var bedre å finne en måte der implementasjonen ville gå raskere. På denne måten kunne vi potensielt utforske flere aspekter ved mekanismen, og forhåpentligvis få en bedre forståelse av hvordan en slik mekanisme ville utspille seg i språket.

Go sitt offisielle språk støtter ikke generiske typeparametere. Det vil si at vi

ikke kunne utnyttet aspekter som ble forklart i 3.5. Nå skal det sies at støtten for generiske typer i GoPT uansett ble implementert separat fra Go sin egen prototype. I starten av prosjektet virket det dog fornuftig å kunne dra nytte av den, noe som ville blitt problematisk ved å implementere konseptene direkte i standardkompilatoren. Derfor var også dette en faktor i beslutningen om å ikke implementere Pakkemal-konsepter i standardkompilatoren.

5.1.2 Uavhengig preprosessor

Siden hovedproblemet med å implementere Pakkemal-konsepter direkte i standardkompilatoren var at det ble for tidkrevende, vurderte vi å lage en preprosessor som ville ta inn ny syntaks for så å skrive om dette til vanlig Go-kode. På denne måten ville vi slippe å bruke mye tid på å forstå eksisterende kode, og vi kunne i tillegg bruke verktøy som var effektive til dette formålet, og som vi også var kjent med fra før. Vi tenkte også at det ikke var så viktig om det resulterte i en fullverdig preprosessor som kunne oversette fullstendige program, men en mekanisme som i alle fall kunne håndtere oversetting av Pakkemal-konsepter til gyldig Go-kode.

For å slippe å utvikle en *parser* på egen hånd helt fra bunnen av, så utforsket vi ulike *yacc*- og *lex*-pakker[1]. Målet var at vi her kunne definere syntaks uten å måtte implementere selve parseren. Dette ville nok til en viss grad vært mulig, men det var likevel mye som ville gjenstå før vi hadde det som trengtes i en preprosessor. Det gjenstående i preprosessoren var også elementer som egentlig ikke ville gitt noe særlig verdi til selve oppgaven. I hovedsak ville dette gått mer på utvikling av en preprosessor, og ikke kombinasjonen av Go og Pakkemal-mekanismen.

Vi vurderte det derfor dit hen at heller ikke dette var en god måte å løse det på.

5.1.3 Videreutvikling av Go-preprosessor

Måten vi endte opp med å implementere Pakkemal-konseptene i Go ble faktisk et slags kompromiss mellom de to nevnte metodene over, der dette gikk ut på å videreutvikle en eksisterende preprosessor. Dette er preprosessoren som ble nevnt i 3.5, og som blir brukt for prototyping av generiske typeparametere i Go. Denne kan vi kjøre ved hjelp av Go sine *tools*, og ligger under en av "prøveprosjekt-grenene" på deres GitHub-konto [28]. Grunnen til at vi kan kalle det et kompromiss, er fordi det både er en preprosessor og at vi gjør endringer i Go sin kildekode.

Denne mekanismen er designet for å prøve ut nye konstruksjoner i språket, der Google selv brukte den for å prøve ut konstruksjoner for generisk typeparameterisering. Det å prøve ut nye konstruksjoner var også det vi ønsket, så derfor tenkte vi at den også virket velegnet til det vi ville gjøre. Forhåpentligvis ville dette føre til at vi nokså enkelt og raskt kunne implementere Pakkemal-konsepter. En konsekvens av dette kunne derfor være at vi ville ha mulighet til å utforske et bredere utvalg av konsepter enn det som ville vært mulig ved hjelp av de andre metodene, og på den måten få en bedre innsikt i hvordan Pakkemal-mekanismen ville utspille seg i Go.

[1] Lex og yacc er verktøy som er designet for rask prototyping av kompilatorer og interpreter med enkle modifikasjoner [18]

Siden formålet med bruken av preprosessoren i all hovedsak var det samme for oss og utviklerne av Go, kunne vi også ta utgangspunkt i hvordan de konkret hadde implementert nye konstruksjoner i preprosessoren. I deres tilfelle brukte de en kombinasjon av ny syntaks og vanlig Go-kode, noe som var akkurat det vi hadde tenkt til å gjøre. Dette var gunstig, fordi ved å følge steg for steg måten de omgjorde den nye syntaksen til vanlig Go-kode, så hadde vi en slags "oppskrift" for hvordan vi kunne gjøre det i vårt prosjekt. Dermed kunne vi raskt starte prosessen med å utvide preprosessoren til å støtte konstruksjoner for Pakkemal-mekanismen.

Som nevnt i 3.4 har Go en rekke pakker i sitt standard-bibliotek. Preprosessoren bruker i stor grad disse pakkene for aspekter som parsing og typesjekking. Dette vil si at mye av arbeidet som måtte til for at preprosessoren skulle støtte Pakkemal-mekanismen handlet om å tilpasse og endre kode i disse pakkene i standard-biblioteket. En av fordelene med det er at Go har laget en rekke tester for disse pakkene, noe som gjorde det veldig enkelt å kjøre regresjonstester etter at vi har endret noe. Dermed fikk vi raskt tilbakemelding om en endring eller utvidelse vi hadde gjort førte til uheldige sideeffekter, og som kanskje ødelagte noe andre steder i kildekoden.

En annen fordel med å videreutvikle denne preprosessoren er at implementasjonen her er nokså lik implementasjonen i standardkompilatoren. Dersom vi i fremtiden vil forsøke å implementere Pakkemal-konseptene i standardkompilatoren, vil vi nok kunne gjøre det på en lignende måte som er gjort i denne oppgaven. Kanskje enda viktigere vil vi kunne avdekke aspekter som vanskeliggjør introduksjon av Pakkemal-mekanismen, og antageligvis i større grad enn hvis vi hadde laget en helt uavhengig preprosessor fra bunnen av. En annen stor fordel med å bruke denne preprosessoren er at vi allerede har støtte for vanlig Go-kode, og kan dermed bruke ressurser på det som faktisk gir verdi.

Prosessen for å introdusere generiske typeparametere i Go er som nevnt tidligere ved å først utvikle en prototype, så diskutere og gjøre endringer før man tilslutt implementerer det i hovedspråket dersom det er konsensus om forslagene. Sånn sett kan vi kanskje si at vi i denne oppgaven følger Go sin foretrukne måte ved å først lage en prototype, i stedet for å direkte forsøke å implementere det i språket.

Selv om det å bruke en preprosessor i denne oppgaven var fornuftig, må det også nevnes at det er noen iboende ulemper med å bruke en preprosessor. Dette kommer av at konseptene ikke blir en del av selve språket som sådan, og at de må oversettes. Bruk av preprosessorer vil føre til et ekstra abstraksjonsnivå for utvikleren, og kan vanskeliggjøre for eksempel feilmeldinger. I tillegg er det også et kjent problem at denne fremgangsmåten kan gi opphav til at semantiske eller implementasjonsmessige kompromisser blir nødvendige.

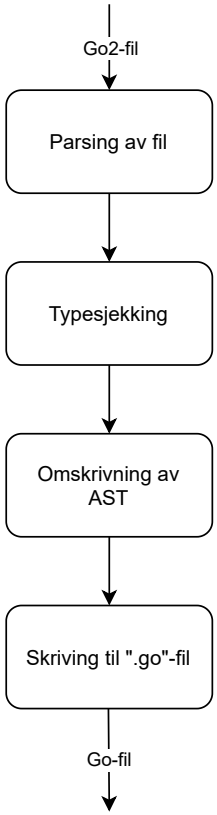
5.2 Preprosessoren i korte trekk

Som vi har vært inne på tidligere så går implementasjonen av GoPT ut på å videreutvikle en eksisterende preprosessor. Formålet med denne preprosessoren som Google har utviklet er å prototype måten generiske typeparametere kan fungere i Go. For å kunne kjøre preprosessoren har de definert et nytt *verktøy* "go2go" som vi kan bruke som kommandolinjeargument. Filer vi vil oversette bruker et nytt filetternavn, ".go2", og kan dermed inkludere ny Go-

syntaks. Under følger et eksempel på kommandolinjeargumenter som oversetter filen `fil.go2`:

```
go tool go2go translate fil.go2
```

Dette vil parse og typesjekke filen og deretter oversette denne til vanlig Go-kode, noe som resulterer i en ny fil "fil.go". Den resulterende Go-filen vil dermed kunne kompileres og kjøres med en vanlig Go-kompilator. Under følger en noe forenklet illustrasjon av hvordan denne transformasjonen skjer:



Figur 5.1: Stegene i Go sin preprosessor

Steget "omskrivning av AST" ("AST" er forkortelse for *abstrakt syntakstre*) er det steget hvor vi hovedsakelig har gjort endringer og tillegg for å kunne støtte Pakkemal-konsepter. I det påfølgende steget, som går på skriving til vanlig Go-fil, var det kun få tillegg i koden som var nødvendig.

Rent praktisk har vi lastet ned kildekoden fra Go sin GitHub-konto. Deretter har vi opprettet vårt eget oppbevaringssted på Bitbucket. Igjennom hele perioden har vi lastet opp kode dit, noe som har vært veldig nyttig siden det for eksempel har tillatt å enkelt gå tilbake til gamle versjoner dersom dette har vært nødvendig.

5.3 Nøkkelord, operatorer og skilletegn

Før vi kan parse et program er vi nødt til å bryte opp det innsendte programmet til ulike *tokens*[2]. Dette er en *leksikalsk analyse* der vi på forhånd har definert hvilke tegn, eller kombinasjoner av tegn, som er gyldig i språket og hva disse dermed skal klassifiseres som. Typisk vil man definere slike regler og klassifiseringer ved bruk av regulære uttrykk.

I preprosessoren var det naturligvis allerede utviklet et slik program. Det som så var nødvendig var å legge til nye nøkkelord og operatorer som ikke var del av språket før introduksjonen av Pakkemal-mekanismen. Måten vi definerte nye nøkkelord og operatorer var veldig enkelt. For å for eksempel legge til `pttemplate` var alt som behøvdes for å få til dette å legge til to linjer med kode. Den ene linjen gikk ut på å legge til en ny konstant `PTTEMPLATE` i en liste for å definere et nytt "token", og den andre linjen var `PTTEMPLATE: ''pttemplate''` som var en måte å beskrive hvordan dette nøkkelordet faktisk så ut i programmet.

Som vi har vært inne på tidligere kan introduksjon av nye nøkkelord ha store sideeffekter. De stedene i koden vi gjør endringer for å få til dette er i pakker fra standard-biblioteket. Disse pakkene blir flittig brukt andre steder i kildekoden til Go, og selvsagt også av "vanlige" brukere av Go. Dersom det tidligere er brukt navn på variabler eller type-deklarasjoner som nå er et nøkkelord i språket, vil dette i de aller fleste tilfeller føre til at programmet ikke lenger er gyldig. Siden det å definere nye nøkkelord kan føre til problemer, og det faktum at Go har bevisst et veldig begrenset antall nøkkelord i språket, så har vi valgt å ikke introdusere særlig mange nye nøkkelord. Under følger en liste over de nye nøkkelordene vi har introdusert:

- `pttemplate`
- `ptinst`
- `addto`
- `toverride`

Fra listen over kan vi se at for eksempel `tsuper` ikke er et nøkkelord, selv om bruken av det kanskje kan se slik ut. I implementasjonen av "tsuper"-konseptet var det faktisk også enklere dersom dette ikke var et nøkkelord, men i stedet en identifikator. Dette var også tilfellet ved implementasjonen av andre konsepter, som for eksempel `with` og `required`. Den kanskje aller største fordelen med å klassifisere nye begreper som identifikatorer, i stedet for å introdusere nye nøkkelord, er at dette ikke fører til sideeffekter andre steder i koden.

Når det gjelder bruk av operatorer og skilletegn, har vi faktisk sluppet å måtte definere nye. I enkelte tilfeller har vi i stedet for å definere nye heller brukt kombinasjoner av tegn som allerede er definert. Et eksempel på dette kan være `=>` som benyttes for endring av navn på for eksempel structer og grensesnitt. Dette er ikke klassifisert som et "token", men vi bruker i stedet de to separate tegnene "=" og ">". Dette er for øvrig noe som må tas hensyn til i parseren, og vi har også tatt høyde for at dette ikke fører til uheldige sideeffekter.

[2]Skanneren deler opp det innsendte programmet og klassifiserer disse små delene av programmet basert på syntaktiske konstruksjoner i språket. Dette resulterer i ulike tokens som består av hva slags klassifisering det er, for eksempel en bestemt datatype eller operator, og dens eventuelle verdi, som for eksempel selve tallet til en `int` [23].

5.4 Parsing

Vi skal nå se på implementasjonen som er gjort på stadiet hvor vi sjekker om det innsendte programmet er syntaktisk gyldig. Parseren som skulle utvides var en variant av "recursive descent" [9]. Dette vil si at parsing av en fil starter med å rekursivt bygge opp et AST der vi ekspanderer ut fra filen, og måten den er definert.

For å beskrive hvordan en "recursive descent"-parser fungerer, kan vi litt forenklet se for oss at en fil består av en samling deklarasjoner. Dette kan vi så anta i parsingen av en fil, og dermed for hver deklarasjon i filen kalle på en metode `parseDecl` helt til vi treffer tegnet som forteller at vi er på slutten av filen. Metoden `parseDecl` vil returnere en AST-node med data om deklarasjonen den har parset, og disse deklarasjonene vil så lagres i en liste i metoden som parser filen. Når vi har truffet tegnet som forteller at vi er på slutten av filen, vil parseren opprette en AST-node for filen og lagre disse deklarasjons-nodene i AST-noden. La oss anta programmet under:

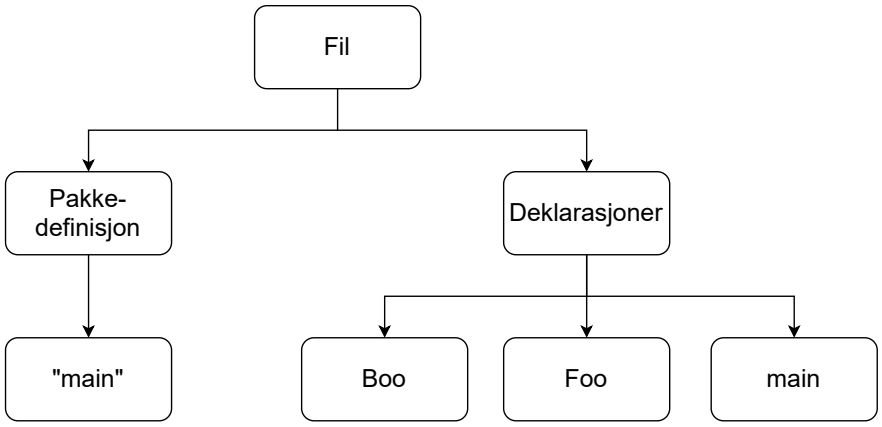
```
package main

const Boo = "boo"

func Foo() string {
   return "foo"
}

func main() {
   ...
}
```

Parsingen av denne filen ville først sjekket om syntaksen som blir brukt for hvilken pakke filen tilhører er korrekt. Siden alle filer i Go krever at pakken den tilhører må spesifiseres, ville parsingen stoppet dersom filen ikke hadde hatt med dette, eller det hadde vært gjort på en ugyldig måte. Videre ville den parset de tre deklarasjonene hver for seg, noe som nevnt gjøres rekursivt. For konstanten `Boo` ville dette først ført til et kall på en generell metode for parsing av deklarasjoner (`parseDecl`). Deretter ville denne metoden funnet ut at det var en konstant, og kalt videre på en metode som spesifikt sjekket om syntaksen her var korrekt. Denne ville igjen kalt på en metode som for eksempel sjekket om verdien til konstanten var syntaktisk gyldig. Når vi så ville returnert fra disse ulike metodene, ville dette bygget opp AST-et gradvis, og vært fullstendig når vi hadde returnert fra parsingen av fil. Programmet over ville resultert i AST som dette:



Figur 5.2: Et forenklet AST for eksempel-programmet over

Som vi allerede har sett i kapittel 4 så vil Pakkemal-konsepter kunne være deklarert i det ytre skopet, noe som i eksempelet over ville vært i samme skop som de tre andre deklarasjonene. For å legge til syntaktiske sjekker for disse konseptene, vil vi i `parseDecl`-metoden legge til funksjonalitet som oppdager at det er Pakkemal-konsepter vi står overfor. Når vi så har funnet hvilket Pakkemal-konsept som blir brukt, for eksempel pakkemal-deklarasjon eller instansiering av en pakkemal, kan vi så kalle på en metode som sjekker om dette er syntaktisk gyldig. Dersom det hadde vært en pakkemal-deklarasjon, ville en metode `parseTemplateDecl` bli kalt, og denne ville returnert en AST-node av typen `TemplateDecl` som tilslutt hadde blitt lagt til i AST-noden til filen.

5.4.1 Pakkemal-deklarasjoner

Pakkemal-deklarasjoner og filer i Go har en del fellestrekk når det kommer til hvordan de er strukturert. Den aller største bestanddelen av filer består av ulike deklarasjoner, noe som også er tilfellet for pakkemal-deklarasjoner. Dette var veldig kurant, fordi syntaktisk sjekk av pakkemaler dermed lignet mye på syntaktisk sjekk av filer, noe som også gjorde at vi kunne gjenbruke en hel del kode. Deklarasjoner som eksempelvis structer og grensesnitt ble enkelt parset ved å kalle på `parseDecl` som videre ville ta seg av dette.

For deklarasjoner og konstruksjoner som vi har introdusert i denne oppgaven, måtte vi naturligvis skrive disse parsing-metodene selv. Disse vil bli forklart videre i kapittelet. Når det kommer til parsing av Pakkemal-konsepter i pakkemal-deklarasjoner, vil det i stor grad handle om å identifisere hva slags konsept det er, og deretter kalle videre på de egendefinerte parsing-metodene for disse.

Parsingen av pakkemal-deklarasjon vil returnere en AST-node som representerer pakkemalen. Dermed måtte vi definere en ny type for dette som inneholdt attributter slik at vi fikk lagret nødvendig data. Dette er hovedsakelig andre AST-noder som representerer deklarasjonene inne i pakkemalen, andre Pakkemal-konsepter som er brukt i den, samt metadata slik som hvor i programfilen pakkemalen befinner seg.

5.4.2 Instansiering av pakkemaler

Det ytre skopet i en Go-fil består i hovedsak av deklarasjoner, og siden instansieringer av pakkemaler kan forekomme her, valgte vi å behandle dette i parsingen som om det var en deklarasjon. Antagelig hadde det vært mer passende å klassifisere det som et "statement", men dette ville gjort implementasjon senere veldig kronglete. Derfor vil en syntaktisk sjekk av dette gjøres på omtrent samme måte som for pakkemal-deklarasjoner. Dette innebærer at vi i `parseDecl` er nødt til å finne ut at det vi skal sjekke er en instansiering av en pakkemal, og videre kalle på en ny metode som sjekker om dette er gjort på en syntaktisk gyldig måte. Metoden for syntaktisk sjekk av pakkemal-instansieringer vil også opprette en ny AST-node der vi lagrer all nødvendig informasjon.

Instansieringer kan inneholde spesifiseringer for endring av navn på typer og typers attributter, funksjons-navn og konkretiseringer av påkrevde typer. I parsingen av instansieringer var dette den største utfordringen, og spesielt det å kunne lagre all nødvendig informasjon om dette på en fornuftig måte. Disse spesifiseringene blir som kjent listet opp i en komma-separert liste, og i parsingen av dette tok vi for oss ett og ett element i listen.

For å lagre informasjonen om endringer av navn på en type og dens attributter, definerte vi en tre-struktur som både tok vare på det nye navnet til selve typen, og samtidig eventuelle nye navn til denne typens attributter. Dersom typen var et grensesnitt, var dette nokså enkelt å håndtere, siden endringer av navn for dette ville innebære selve navnet på grensesnittet eller dens metodesignaturer. I den komma-separerte listen over endring av navn, vil endring av grensesnitt og dets attributter bli angitt i samme element i listen, for eksempel slik:

```
Grsnittnavn => NyttGrsnittNavn(Metodenavn -> NyttMetodenavn)
```

Dette gjorde at vi fikk lagret all informasjon samtidig, noe som var forholdsvis gunstig implementasjonsmessig. Etter å ha parset hele elementet i den komma-separerte listen, kunne vi dermed opprette et objekt av tre-strukturen som ville inneholdt det opprinnelige og nye grensesnitt-navnet, samt det opprinnelige og nye metode-navnet.

Når det kommer til structer kompliseres dette litt, fordi endring av navn på structen og structens felter spesifiseres samtidig, mens endring av metoder som er tilknyttet structen vil skje i et annet element i den komma-separerte listen. Endring av navn på en struct og dens metoder spesifiseres altså ved ulike elementer i listen. Dette gjør, i motsetning til når vi håndterer grensesnitt, at vi ikke får lagret all informasjon om endringer av navn for en struct samtidig. Kombinasjonen av dette og at vi ikke vet i hvilken rekkefølge i listen disse spesifikasjonene kommer, for eksempel om endringen av struct-navnet kommer før metode-navnet, gjør at vi er nødt til å legge til en rekke sjekker og ta høyde for diverse grensetilfeller.

Det å lagre endring av navn på vanlige funksjoner og konkretiseringer av påkrevde typer var ikke like utfordrende som håndteringen av structer. Disse har begge unik syntaks, og etter vi har parset og lagret informasjonen fra disse spesifikasjonene, ville vi ikke trenge å endre denne informasjonen senere. For øvrig var vi nødt til å definere en ny datastruktur for konkretisering av påkrevde typer, noe som kun innebar å kunne lagre navnet på den påkrevde typen og navnet på den konkrete typen.

5.4.3 "Addto"-setningen

Syntaksen som er definert for "addto"-setningen gjør at parsingen av dette kan gjøres på en enkel måte. Vi kan bruke setningen for tre ulike typer, henholdsvis structer, grensesnitt og påkrevde typer. Når vi vil utvide en av disse, brukes deres respektive nøkkelord foran navnet på typen, slik at vi på denne måten vet hva slags type som skal parses før vi parser kroppen. Dette er gunstig, fordi det tillater å kunne gjenbruke kode i en veldig stor grad.

Dersom det er en påkrevd type som skal utvides, kan vi kalle direkte på parsing-metoden for deklarasjon av påkrevde typer. Siden dette er et nytt konsept som vi har introdusert i denne oppgaven, har vi måttet skrive denne selv. Dog er innholdet av en slik type definert på omtrent lik måte som innholdet av grensesnitt, slik at dette innebar i prinsippet å gjøre det på tilsvarende måte. Naturligvis måtte vi definere en ny AST-node tilpasset for påkrevde typer.

Hvis det var et grensesnitt som skulle utvides, kunne vi bruke parsingen for innhold av grensesnitt direkte. For structer var vi nødt til å skille mellom nye felter og metode-deklarasjoner. Dog var alt som krevdes for dette en "if"-setning, og vi kunne videre kalle på parsing-metoder som allerede var definert. For metode-deklarasjoner måtte vi passe på om dette var en overskriving av en annen, altså om metode-deklarasjonen var markert med nøkkelordet `toverride`. Dette var fordi vi i AST-noden til "addto"-setningen ønsket å lagre alle overskrevne metoder separat.

Overskriving av vanlig funksjoner gjøres som nevnt tidligere ved å bruke notasjonen `addto _ { ... }`. Vi vil altså her ikke spesifisere hva slags type vi skal utvide, men bruke understrek i stedet. Dog trengte vi ikke å håndtere dette på en annerledes måte enn for metoder som er tilknyttet en struct, slik at parsingen av dette kunne gjøres på akkurat lik måte.

5.5 Den første typesjekken

Etter en vellykket syntaktisk analyse, vil vi ende opp med et komplett AST som representerer det innsendte programmet. Det neste vi nå ønsker å gjøre er å bruke dette treet for å sjekke om semantikken i programmet er gyldig. AST-et inneholder nå flere nye node-typer som det naturligvis ikke er implementert støtte for enda. Derfor vil neste steg være å støtte typesjekking av disse, og gjøre en semantisk analyse.

Slik preprosessoren opprinnelig var definert, altså før vi gjorde endringer og utvidelser i den, kunne vi være sikker på at programmet var semantisk gyldig etter denne typesjekken. Det vil si at etter dette så trengte vi ikke å gjøre flere semantiske analyser på et senere tidspunkt. Dersom vi skulle implementert en fullstendig semantisk analyse av Pakkemal-konsepter på dette tidspunktet, ville koden for dette blitt veldig kompleks og komplisert. Vi fant derfor ut at det ville være fornuftig å gjøre typesjekking av disse konseptene i flere omganger. På denne måten var det enklere å skrive kode som kunne forsikre om at det resulterende programmet ville være semantisk gyldig.

I denne delen vil vi se på den første typesjekken der formålet hovedsakelig er å sjekke at pakkemalene i et program isolert sett er gyldige. Det vil si at vi nå ikke tar hensyn til hvordan programmet ender opp etter instansieringer og tilpasninger av pakkemaler. Et eksempel på dette kan være at det i skopet på

utsiden av pakkemalene er deklarert typer med samme navn som i pakkemalen, og som til og med kanskje blir instansiert uten endring av navn, noe som likevel skal passere i denne sjekken dersom pakkemalene er gyldig som en isolert enhet.

Det at vi kan typesjekke pakkemal-deklarasjoner i isolasjon er et ganske viktig poeng med Pakkemal-mekanismen. For det første ville det vært veldig ugunstig å ikke kunne vite om en pakkemal-deklarasjon er typesikker før etter instansiering, siden det på denne måten kan være vanskeligere å spore tilbake til hva som faktisk er ugyldig. For det andre kan vi tenke oss scenarioer der vi ønsker å deklarere pakkemaler uten å instansiere de i samme program. Uten å kunne typesjekke pakkemaler i isolasjon, ville vi ikke kunne vite om programmet med pakkemal-deklarasjonene (uten instansieringer) faktisk var typesikker.

Spesifikke semantiske analyser som går utover deklarasjoner av pakkemaler vil bli grundigere tatt hånd om når vi modifiserer AST-et, og blir derfor forklart i 5.6. En komplett typesjekk av det resulterende programmet skjer etter at alle modifikasjoner på AST-et er gjort, og vil bli forklart i 5.7.

5.5.1 Typesjekking av pakkemaler

Siden innholdet i en pakkemal hovedsakelig vil være konsepter som allerede er definert i Go, ønsket vi å forsøke å gjenbruke kode som allerede var skrevet for typesjekking av dette. Vi har lagt nokså få begrensninger for hva som er tillatt å definere inne i pakkemaler, og i praksis vil vi kunne deklarere og skrive den samme koden inne i pakkemaler som det som er mulig utenfor. Det å skrive kode for komplett typesjekking av dette ville nesten vært en masteroppgave i seg selv, og vi var derfor veldig opptatt av å kunne gjenbruke mest mulig kode. Siden denne første typesjekken av AST-et hovedsakelig skal sjekke at deklarasjoner av pakkemaler er semantisk gyldig, kunne vi kanskje tenke at det beste ville være å eksplisitt kun sjekke disse. Det er noe som absolutt ville vært mulig, men det hadde dog krevd en god del mer kode-skriving, noe vi fant ut at det ikke var tid til med tanke på tidsaspektet for en slik oppgave.

Som nevnt i 5.4.1 har filer og pakkemaler en rekke fellestrekk, noe som betyr at typesjekken også vil kunne ligne. Det vi derfor valgte å gjøre var å forsøke å gjenbruke store deler av typesjekkingen som var gjort for filer, for typesjekking av pakkemaler. Dette ville føre til at vi nå ville slippe å implementere typesjekking for konsepter som allerede var definert i Go.

Den store utfordringen knyttet til å gjenbruke typesjekkingen for filer var dette med å håndtere skop. I samme fil som en pakkemal-deklarasjon befinner seg kan det jo nemlig også være en rekke andre deklarasjoner. Dette er dog noe som ikke skal ha noen innvirkning på typesjekkingen av en pakkemal, noe som var vanskelig på grunn av måten dette var implementert. Et eksempel kan være at det er definert en struct `S` i det ytre skopet, og at det i en pakkemal er definert en struct med samme navn. Dette er noe som skal være tillatt på dette stadiet, fordi vi for eksempel i en instansiering kan endre navnet på structen i pakkemalen slik at det resulterende programmet ender opp gyldig. Uten å eksplisitt håndtere skop for pakkemaler, ville typesjekkeren ment at deklarasjonen av structen i pakkemalen ville vært en "redeklarasjon", og avsluttet med en feilmelding.

Vi forsøkte først å løse dette ved å lage nye AST-noder for pakkemalene der disse nodene ville for typesjekkeren se ut som filer. På denne måten ville vi ikke lenger bli påvirket av andre deklarasjoner som var definert i skopet utenfor, og vi ville samtidig fått gjenbrukt en veldig stor del av typesjekkingen. Dog skulle

det vise seg at det å lage disse nye nodene, og samtidig få til en fullstendig typesjekk av pakkemalene uten at dette førte ved seg uheldige sideeffekter, ville bli en stor utfordring. I tillegg ville det muligens komplisere omskrivningen av AST-et på et senere tidspunkt.

Måten vi i stedet valgte å løse det på var ved å skrive om alle deklarasjonsnavn i pakkemalene til nye, "unike" navn. På denne måten ville vi nå unngå å bli påvirket av deklarasjoner i skopet på utsiden av pakkemalene. I eksempelet der vi hadde deklarert en struct `S` både i det ytre skopet og i en pakkemal, ville ikke dette lenger ført til en redeklarasjons-feil. Typesjekkeren ville i stedet nå sett to forskjellige structer som ikke hadde noen sammenheng. Måten vi løste dette på rent teknisk var ved å gi alle navn et prefiks før selve typesjekkingen. Dette er noe som fungerte godt i denne prototypen, men i en reell implementasjon vil det nok være fordelaktig å gjøre det på en annen måte, siden det å endre navn i utgangspunktet er noe som først krever en typesjekk for å være sikker på at endringene blir riktig. Implementasjonen av dette innebar å traversere AST-et etter fullført parsing, og legge til et prefiks (forskjellig for hver pakkemal) på alle deklarasjonsnavn i pakkemaler, samt alle stedene disse ble brukt. Disse prefiksene var kun midlertidige slik at før vi transformerte AST-et til vanlig Go-kode, fjernet vi prefiksene.

Grunnen til at vi i starten på forrige avsnitt markerte ordet "unike" med anførselstegn, er fordi det ikke er en garanti i denne implementasjonen at navnene faktisk blir unike. Dette er dog noe man burde sørge for i en reell implementasjon, og som nokså enkelt kunne vært løst med for eksempel bruk av en *UUID-generator*[3]. Slike generatorer finnes i en rekke eksterne pakker, og ville vært lettvint å importere og benytte seg av. I denne oppgaven ble ikke dette prioritert.

Det å legge til disse prefiksene løste problemet med skop, og vi kunne dermed gjenbruke det meste av typesjekkingen som var definert for filer. Dette gjorde at pakkemalene nå ble typesjekket som om de var i isolasjon. Dermed ville vi få feilmeldinger med nyttig informasjon dersom det fantes typefeil i kode som var skrevet inne i en pakkemal.

5.5.2 Typesjekking av de øvrige Pakkemal-konseptene

For typesjekking av de øvrige Pakkemal-konseptene, som for eksempel "ptinst"- og "addto"-setningene, har vi ikke implementert dette på tilsvarende måte som for pakkemal-deklarasjoner. Det å for eksempel skulle typesjekke en "addto"-setning på dette tidspunktet ville vært ganske komplekst. Dersom vi skulle gjort dette, måtte vi tatt hensyn til andre Pakkemal-konsepter og pakkemalen selv. Det som avgjør om en "addto"-setning er semantisk gyldig avhenger av mange andre faktorer. Derfor fant vi ut at det ville være bedre å gjøre en komplett typesjekk senere etter at instansieringer og tilpasninger av pakkemaler var gjort.

Det å ikke typesjekke disse konseptene eksplisitt, kan dog føre til dårligere feilmeldinger. Derfor har vi også implementert en rekke sjekker etter denne første typesjekken. Dette foregår for det meste samtidig som vi modifiserer AST-et, og inkluderer aspekter som for eksempel å sjekke om angitte overskrivinger av funksjoner er gyldig i "addto"-setninger. Vi kan derfor si at typesjekking av de

[3]En UUID-generator vil lage et 128-bits langt nummer, der sjansen for å generere et nummer som allerede finnes er neglisjerbar [17]

øvrige Pakkemal-konseptene i hovedsak skjer på et senere tidspunkt, og vil bli forklart i 5.6.

5.5.3 Aksesseringer av Pakkemal-konsepter

Denne første typesjekken, der formålet hovedsakelig er å sjekke om pakkemaler i isolasjon er semantisk gyldig, skjer som nevnt før vi har instansiert pakkemalene. På grunn av dette vil særlig aksesseringer av innhold i pakkemaler føre til feil i typesjekkingen. Dette er hovedsakelig fordi det som skal aksesseres ikke er synlig for typesjekkeren før etter instansiering av pakkemalene er håndtert. For at ikke typesjekken skal feile og avslutte som en følge av dette, har vi lagt inn en del eksplisitte sjekker. Vi vil under forklare et utvalg av dem:

- **Endring av navn:** Dersom vi instansierer en pakkemal og tilpasser denne ved å endre navn, vil dette kunne føre til problemer dersom vi aksesserer dette som har endret navn. For typesjekkeren vil det se ut som vi forsøker å aksessere noe som ikke er deklarert. Derfor har vi lagt inn en sjekk som hindrer at typesjekkingen avsluttes dersom den finner en instansiert pakkemal der vi endrer navn til det samme som tilsynelatende ikke er deklarert.

- **Utvidelse av pakkemal:** Ved å bruke "addto"-setningen vil vi også potensielt kalle på noe som for typesjekkeren ikke er deklarert. Under denne første typesjekken vil ikke dette være synlig for typesjekkeren, slik at dette ser ut til å ikke eksistere. Derfor vil vi være nødt til å eksplisitt sjekke om det vi aksesserer er definert i en "addto"-setning.

- **Påkrevde typer:** En variabel typet som en påkrevd type vil ofte kunne resultere i en "invalid type" for typesjekkeren. Andre egendefinerte typer som for eksempel structer og grensesnitt lagres i ulike interne lister og tabeller, noe vi ikke kan gjøre for påkrevde typer. Derfor har vi heller utviklet sjekker som kan finne ut om det som tilsynelatende er ugyldig type faktisk er en påkrevd type.

  Dersom vi kaller på metoder som via en variabel er typet som en påkrevd type, vil vi også kunne ende opp i situasjoner der typesjekkeren mener at typen ikke har metoden definert. Derfor vil vi også måtte håndtere dette.

Det er også noen andre spesifikke tilfeller vi er nødt til å håndtere, og fellesnevneren for disse sjekkene er at vi er nødt til å traversere AST-et og sjekke om dette faktisk er lovlig som følge av Pakkemal-konsepter. Etter hvert i utviklingsprosessen, la vi til et attributt i typesjekkeren som var sant dersom den var inne i den første typesjekken. Dette var ganske gunstig, fordi vi kunne dermed droppe å skrive veldig komplekse sjekker i AST-et, og heller sjekke det samme i den andre typesjekken. Endringer av AST-et, som følge av bruk av Pakkemal-konsepter, kan jo nemlig ha ført til at det som tilsynelatende var ugyldig kode faktisk har ført til gyldig kode. Dersom dette ikke er tilfellet, vil uansett den andre typesjekken plukke opp feilene, og på dette tidspunktet avslutte preprosessering og skrive ut en feilmelding.

5.6 Modifikasjoner på AST

Etter den første typesjekken vil neste steg være å gjøre modifikasjoner på AST-et, for å transformere et "GoPT-AST" til et vanlig "Go-AST". Dette innebærer å håndtere Pakkemal-konseptene som å blant annet instansiere og tilpasse pakkemaler. I tillegg vil vi på dette steget sjekke semantiske aspekter ved Pakkemal-konsepter. En av grunnene til at en del av disse sjekkene kommer på akkurat dette tidspunktet, er fordi det gjør det enklere å skrive presise feilmeldinger.

Det preprosessoren opprinnelig gjorde på dette steget, var å oversette generisk typeparameter-syntaks til vanlig Go-syntaks. Derfor tenkte vi at dette var et passende sted å implementere selve støtten for Pakkemal-konsepter (utover det å kun godta syntaks). Det å håndtere disse konseptene på et AST-nivå er også gunstig, siden det finnes en del funksjonalitet for blant annet håndtering av AST-noder som vi kunne benytte oss av.

5.6.1 Semantiske sjekker før modifikasjoner

Før vi faktisk modifiserer AST-et, vil vi kjøre noen sjekker som sørger for at semantikken vi har definert for GoPT overholdes. Vi vil nå beskrive disse.

Endring av navn på grensesnitts metode-signaturer

Det å endre navn på metoder som er definert i et grensesnitt, kan føre til problemer. Dette kommer av Go sin strukturelle typesjekking for grensesnitt, som tillater at instanser av en struct kan types som et grensesnitt dersom den har definert alle metodene som grensesnittet har. For å eksemplifisere dette problemet kan vi anta følge pakkemal:

```
pttemplate T {
   type I interface {
      F()
   }

   type S struct {
      s string
   }

   func (s S) F() {
      ...
   }
}
```

I eksempelet over kan vi se at et objekt av `S` kan types til grensesnittet `I`, siden den har definert metoden `F`. La oss videre anta at `T` nå blir instansiert med følgende setning:

```
ptinst T with I => I(F -> NyF)
```

I setningen over instansieres `T`, og vi endrer så metoden `F` i `I` til å nå hete "NyF". Siden det ikke endres navn på structen `S` sin metode `F`, kan ikke objekter av denne structen lenger types som `I`. Dersom det et sted i koden finnes slike forekomster, vil dette føre til problemer, siden dette ikke lenger er gyldig. Med

andre ord kan denne endringen av navn føre til at pakkemalen som opprinnelig var gyldig nå inneholder semantisk ugyldig kode.

Vi vurderte derfor om vi burde forby endring av metode-navn i grensesnitt, noe som ville løst dette problemet. Dog er som nevnt formålet med Pakkemal-mekanismen at den skal være fleksibel, og en slik regel ville gjort den mindre fleksibel. Vi fant derfor ut at det var bedre å tillate dette, og at utvikleren selv må passe på at dette ikke fører til utilsiktede sideeffekter. Det skal også sies at selv om vi tillater dette, så vil det fortsatt ikke være mulig å kompilere et program som ikke er typesikkert. Dersom en endring av et metode-navn fører til at vi forsøker å tilegne et objekt til en variabel av en type som objektet ikke implementerer, vil preprosesseringen avslutte med en feilmelding.

For å hjelpe utviklere å være klar over tilfeller som kan føre til sideeffekter, definerte vi en eksplisitt sjekk på dette som skriver ut en advarsel dersom metode-navn i grensesnitt endres og ikke metoder tilhørende structer med samme signatur. Eksempelet over ville resultert i en advarsel om at vi ikke har gitt structen `S` sin metode `F` det samme navnet som grensesnittet sin metode, og at dette dermed potensielt kan føre til problemer.

Endring av navn på structers metode-deklarasjoner

Problemer lignende det som ble beskrevet i forrige avsnitt, vil vi også kunne støte på dersom vi kun endrer navn på en struct sin metode. La oss anta samme pakkemal `T` som tidligere, men at vi nå endrer navn på `S` sin metode `F`:

```
pttemplate T {
   type I interface {
      F()
   }

   type S struct {
      s string
   }

   func (s S) F() {
      ...
   }
}
ptinst T with S.F => NyF
```

Etter denne instansieringen implementerer ikke lenger structen `S` grensesnittet `I`. På samme måte som tidligere vil dette dermed føre til problemer dersom vi et sted i koden har tilegnet et objekt av `S` til en variabel av `I`. Dersom vi derimot også hadde endret navnet på grensesnittet sin metode `F` til "NyF", ville det derimot fortsatt vært gyldig.

Vi definerte derfor en sjekk som lette etter scenarioer som i eksempelet over, og som i eksempelet ville resultert i en advarsel hvor det ville stått at grensesnitt-metoden `F` ikke ble endret til samme navn som `S` sin metode.

Konkretisering av påkrevde typer

Vi har også laget en sjekk som undersøker om angitte konkretiseringer av påkrevde typer i en "ptinst"-setning er gyldige. Dette innebærer først å sjekke om

vi refererer til en faktisk påkrevd type i pakkemalen. Dersom den påkrevde typen ikke finnes i pakkemalen vi skal instansiere, vil preprosesseringen avslutte, og det vil bli skrevet ut en feilmelding om at den påkrevde typen vi refererer til ikke eksisterer.

Hvis den påkrevde typen finnes, vil vi videre sjekke om den konkrete typen oppfyller kravene som er definert. Det som menes med dette er at den konkrete typen må ha definert alle metoder som er spesifisert i den påkrevde typen. Siden vi per nå kun tillater å konkretisere med structer, så innebærer denne sjekken å finne structen og sjekke at den har definert alle metoder. Dersom dette ikke er tilfellet, vil preprosesseringen avslutte og skrive ut en feilmelding.

Det å sjekke dette såpass tidlig i prosessen viste seg å være veldig gunstig. Blant annet bidrar dette til at vi nokså enkelt kan skrive veldig presise feilmeldinger, der vi enkelt forstår hva som fører til semantiske feil. Dersom vi skulle håndtert disse aspektene etter instansiering, ville det blitt vanskelig for preprosessoren å finne ut av hva som faktisk førte til at semantikken ble ugyldig. Når vi senere skal instansiere pakkemaler, er det også fordelaktig å vite at disse er gyldige, slik at vi slipper å håndtere slike aspekter på det tidspunktet.

I og med at vi tillater en ganske fleksibel bruk av påkrevde typer, så gjør dette at denne sjekken blir nokså kompleks. Vi ønsker selvsagt at den skal fange opp all ugyldig bruk, samtidig som den tillater all gyldig bruk. Det er mange ulike måter å bruke påkrevde typer på, som for eksempel bruk av det inne i andre pakkemaler, endring av navn på krav og videre utvidelse av skranker. Siden sjekken for konkretiseringer av påkrevde typer er på et tidspunkt før selve instansieringene, kan det også være vanskelig å se om bruk av påkrevde typer faktisk ender opp gyldig i det resulterende programmet. For å illustrere dette kan vi anta følgende program:

```
pttemplate T1 {
   required type R {
      S() int
   }
}

pttemplate T2 {
   ptinst T1 with R => G(S -> L)
   addto required type G {
      K() string
   }
}

type S struct {
   s string
}

func (s S) L() int {
   return 1
}

func (s S) K() string {
   return s.s
}
```

```
ptinst T2 with G <= S
```

I programmet over har vi en pakkemal `T1` som har definert en påkrevd type `R`. Denne pakkemalen instansieres i en annen pakkemal `T2`, og navnet på både den påkrevde typen og metoden som er definert i denne blir endret. I tillegg utvides skranken til den påkrevde typen med en ny metode. I det ytre skopet instansieres `T2`, og vi konkretiserer `G` med structen `S`.

Konkretiseringen som skjer i det ytre skopet er i utgangspunktet typen som ble definert i `T1`. Vi kan likevel se at denne typen har endret seg nokså mye fra dette, og programmet ender opp gyldig. Koden vi skrev for sjekking av semantikk på dette ble derfor fort nokså kompleks, siden det er såpass mye forskjellig vi er nødt til å ta høyde for. Vi må også sjekke noen like aspekter flere ganger, og koden bar nok litt preg av dette i form av at det ble litt duplisering av kode som gjorde det samme.

5.6.2 Nødvendige modifikasjoner av AST før instansieringer

Før vi kan sette i gang med selve instansieringer av pakkemaler, er det noen andre modifikasjoner på AST-et som vi er nødt til å gjøre.

Utvidelse av påkrevde typer

I 5.6.1 brukte vi "addto"-setningen for å utvide en påkrevd type med enda flere krav. Disse utvidelsene ønsker vi å gjøre før vi instansierer selve pakkemalene og håndterer andre typer utvidelser.

Måten dette gjøres på er ved å traversere AST-et, og finne alle pakkemaldeklarasjoner som utvider påkrevde typer. I AST-noden til "addto"-setninger har vi en eksplisitt liste for utvidelser av påkrevde typer, slik at det å finne disse er nokså enkelt. Når vi så har funnet en AST-node til utvidelse av en påkrevd type, må vi finne pakkemalen der denne påkrevde typen ble definert. Etter denne og selve deklarasjonen av den påkrevde typen er funnet, kan vi lage en ny AST-node for påkrevd type og sammenflette innholdet i "addto"-setningen med det som opprinnelig var definert for den påkrevde typen. Deretter lagres denne noden som en påkrevd type for pakkemalen hvor utvidelsen fant sted.

Siden vi nå har oppdatert kravene til påkrevde typer, vil vi igjen kalle på sjekken som ble forklart i 5.6.1. Etter denne sjekken vil vi være sikre på at påkrevde typer og konkretiseringer av disse følger de semantiske reglene vi har definert.

Finne hvilken pakkemal en utvidelse tilhører

Da støtten for instansieringer og håndtering av "addto"-setningen ble implementert, var syntaksen definert slik at vi spesifiserte hvilken pakkemal som en struct eller et grensesnitt tilhørte når vi brukte "addto"-setning på disse. Syntaksen var altså slik: `addto PakkemalNavn.TypeNavn { ... }`. På denne måten var det enkelt finne deklarasjonen av typen som skulle utvides, siden vi visste i hvilken pakkemal den befant seg. Dog fant vi senere ut at en mer brukervennlig syntaks ville være å ikke måtte spesifisere pakkemalen der typen var deklarert. Dersom vi for eksempel ville sammenflette typer fra flere ulike pakkemaler, og samtidig

bruke "addto"-setningen på dette, ville det muligens være uklart hvilken pakkemal vi skulle spesifisere. Derfor endret vi syntaksen slik at vi ikke ville trenge å spesifisere pakkemalen.

Denne endringen av syntaks førte dog til at vi nå måtte finne ut hvilken pakkemal den tilhørte selv, og legge til dette som et attributt i AST-noden til "addto"-setningen. Derfor var vi nødt til å traversere AST-et, og første steg var da å finne alle "ptinst"-setninger for å vite hvilke pakkemaler som var aktuelle. I tillegg måtte vi sjekke om disse hadde endringer av navn, siden dette ville føre til at vi var nødt til å lete etter det opprinnelige navnet i pakkemalene, og altså ikke navnet som var angitt i "addto"-setningen. Etter å ha funnet en deklarasjon med riktig navn, kunne vi lagre navnet på pakkemalen som deklarasjonen befant seg i, og terminere søket for denne utvidelsen.

Omskrivning av `tsuper`

I 4.7 viste vi hvordan vi kunne overskrive metoder og funksjoner i en pakkemal, samt muligheten for å fortsatt kalle på den opprinnelige funksjonen. Syntaksen som er definert for kall på en opprinnelig funksjon er slik:

```
tsuper.OpprinneligFunksjon()
```

Dersom vi overskriver en funksjon `F`, skal kallene på `F` etter dette i stedet referere til den nye funksjonen (med mindre notasjonen over blir brukt selvsagt). Dette kan bli løst nærmest av seg selv ved å enten fjerne eller endre navn på den opprinnelige funksjonen. Siden vi skal kunne kalle på den opprinnelige funksjonen, vil jo naturligvis det å fjerne den være dumt, så vi valgte derfor å gi den et nytt navn. For at dette ikke skal føre med seg noen uheldige sideeffekter, burde dette nye navnet være unikt. Dette har vi dog ikke tatt høyde for i denne implementasjonen, slik at det kan være unntaksvise scenarioer der vi får endret navn til noe som allerede er brukt. I en reell implementasjon burde vi ta høyde for dette, og vi kunne også her brukt `UUID`-generatorer som ville løst dette på en nokså enkel måte.

Endringene av navn på funksjoner som blir overskrevet skjer når vi håndterer "addto"-setningen for structer, så dette blir nærmere forklart senere. Det vi dog kan gjøre på dette tidspunktet er å skrive om kallene på den opprinnelige funksjonen. Dette vil si å skrive om notasjonen som bruker ordet `tsuper` etterfulgt av navnet på funksjon, til å nå heller kalle direkte på funksjonen ved å bruke det nye navnet.

Måten dette gjøres på er igjen ved å traversere AST-et, og lete etter konstruksjonen "`tsuper.Funksjon()`". For å lete etter slike konstruksjoner kunne vi bruke funksjonalitet fra en hjelpepakke som senere skal forklares. Ved funn av en slik konstruksjon kunne vi videre sjekke om det både var brukt `tsuper` og at funksjons-navnet var et som skulle overskrives. Dersom dette var tilfellet, kunne vi simpelthen bytte ut hele noden som representerte "tsuper"-kallet med en identifikator-node som representerte det nye funksjons-navnet.

Vi nevnte i 5.3 at vi har vært forsiktig med å definere nye nøkkelord i språket, og at vi ved flere anledninger tillot parseren å heller tolke typiske nøkkelord som identifikatorer i stedet. I dette tilfellet var dette veldig nyttig, siden vi ikke kunne benyttet oss av hjelpepakken dersom `tsuper` var definert som et nøkkelord. Dermed måtte vi ha implementert AST-traversering selv, noe som

antageligvis ville vært komplekst og tatt lang tid å implementere.

5.6.3 Hjelpepakker

Som nevnt tidligere har vi i denne oppgaven kunnet benytte oss av ulike pakker, både i og utenfor standardbiblioteket. Dette har bidratt til at vi har kunnet ha en nokså jevn og god progresjon igjennom prosjektet, som en følge av at vi har sluppet å implementere aspekter som ikke ville gitt noe direkte verdi til oppgaven, og som ville tatt lang tid å implementere. Vi vil nå forklare kort tre ulike pakker som har bidratt til enklere håndtering av AST-et.

Standardbibliotek-pakken "ast"

Denne pakken ble flittig brukt av preprosessoren før vi begynte med våre endringer, og pakken har derfor hatt en stor innvirkning for hvordan programmet er utformet. Allerede ved parsingen utvidet vi denne til å også kunne håndtere Pakkemal-konspeter. Da vi implementerte aspekter som instansiering av pakkemaler og sjekking av semantikk, viste denne pakken seg å ha en del funksjonalitet som vi kunne benytte oss av. Det vi brukte aller mest var en funksjon kalt `Inspect` som tilbød traversering av AST-er på en "dybde-først"-måte. Kanskje aller viktigst gjorde dette at vi ikke trengte å implementere dette selv, noe som sparte oss for masse tid. I tillegg var dette en høyere-ordens funksjon, slik at vi kunne dynamisk sende inn funksjoner, noe som gjorde den fleksibel. Pakken støttet også funksjonalitet som utskrift av trær, noe som var veldig nyttig underveis i utviklingsprosessen.

Astutil

Denne pakken ble i likhet med den forrige hovedsakelig brukt for traversering av AST. Det som skilte denne fra pakken `ast` var at den hadde litt mer funksjonalitet innebygget. For eksempel kunne vi kalle på funksjoner som slettet, byttet ut og satt inn nye noder. Med funksjonaliteten denne pakken tilbød så virket den helt utmerket i starten av prosjektet, men etter hvert som vi definerte nye typer AST-noder bød dette på problemer. Dette kom av at pakken ikke kunne håndtere disse, og siden den ikke var definert i standardbiblioteket var det også litt problematisk å tilpasse den. Det førte til at vi ikke fikk utnyttet all funksjonalitet som var utviklet i pakken. Likevel kom den til nytte ved for eksempel endringer av navn ved instansieringer av pakkemaler.

Astcopy

Et viktig prinsipp som gjelder når vi skal instansiere en pakkemal er at vi kopierer innholdet dens. Når vi skal håndtere instansieringer på "AST-nivå", betyr dette i praksis å dyp-kopiere trær. Dette er også noe som nødvendigvis ikke er helt trivielt å implementere, og som tar lang tid. Derfor var det veldig gunstig at vi fant en pakke som kunne tilby dette.

I likhet med pakken `Astutil` kunne ikke denne håndtere de nye node-typene som vi hadde definert i dette prosjektet. Dermed krevde bruk av denne at vi passet på at trærne vi ville kopiere ikke inneholdt noen nye node-typer. Dersom vi sørget for dette, ville vi kunne dyp-kopiere sub-trær i et AST.

5.6.4 Instansiering av pakkemaler

Som nevnt tidligere kan pakkemaler instansieres både inne i andre pakkemaler og i det ytre skopet. Måten vi har løst dette på er at vi først gjør ferdig alle instansieringer som forekommer inne i andre pakkemaler. Dette inkluderer også videre utvidelser av pakkemalen og andre Pakkemal-konsepter. Etter at all bruk av Pakkemal-mekanismen inne i andre pakkemaler er håndtert, kan vi gjøre det på tilsvarende måte i det ytre skopet. Instansieringene skjer dermed i to omganger, men vi har i stor grad klart å bruke den samme koden for begge tilfellene.

Før håndteringen av selve instansieringene, har vi på forhånd traversert AST-et og funnet bruk av Pakkemal-konsepter, og lagret disse i separate lister. Dette gjør at vi kan iterere en liste bestående av alle instansieringer som har funnet sted inne i andre pakkemaler. For hver instansiering vil vi så bruke pakken `astcopy` og kopiere innholdet til pakkemalen som instansieres. AST-noden for pakkemaler er strukturert slik at vi enkelt vil kunne forsikre oss om at det kun er vanlige AST-noder, og ikke nye definert for dette prosjektet, slik at pakken ikke vil feile som følge av dette. Når vi har det kopierte innholdet av pakkemalen, kan vi videre gjøre spesifiserte tilpasninger direkte uten at dette vil føre med seg uønskede sideeffekter.

Prefiksene vi la til før typesjekking, for å kunne gjenbruke eksisterende kode, er noe vi også må ta høyde for her. Disse er på dette tidspunktet fortsatt ikke fjernet, og det kopierte innholdet vil ha et prefiks som er spesifikt for pakkemalen vi instansierer. Dette må nå endres slik at prefikset i stedet er det samme som pakkemalen der instansieringen finner sted. Å finne alle disse deklarasjonene og identifikatorene som har et slikt prefiks er en nokså krevende prosess. Dog var pakken `astutil` nyttig å bruke her, siden vi dermed kunne abstrahere bort en del av de spesifikke detaljene knyttet til traversering.

Etter dette var tatt hånd om, ville neste steg være å ta høyde for spesifiserte endringer av navn.

Endring av navn

Det å ta høyde for endring av navn i en pakkemal er noe av det mest komplekse i denne oppgaven, og implementasjonen av dette har krevd flere hundre kodelinjer. I tillegg til det å konkret endre navn, er det også diverse semantiske aspekter som vi er nødt til å håndtere.

Under parsingen lagret vi alle spesifiserte endringer av navn på en strukturert måte. Vi vil derfor nå gå igjennom hver og en av disse. Det første vi leter etter er selve deklarasjonen på det vi skal endre navn på. Når vi så har funnet riktig deklarasjon, vil vi endre alle nødvendige noder slik at vi nå i stedet bruker det nye navnet.

Etter vi har funnet deklarasjonen og endret de nødvendige nodene tilknyttet deklarasjonen, vil vi også bli nødt til å endre bruken av det gamle navnet. Dette innebærer å traversere hele pakkemalen og lete etter det gamle navnet. Dersom vi finner det gamle navnet, vil dette endres til det nye. For dette kunne vi igjen benytte oss av en hjelpepakke, noe som forenklet denne prosessen.

Når vi endrer navn på typer og attributter i pakkemaler, er disse endringene som nevnt basert på semantiske bindinger. Derfor holder det ikke å kun gå igjennom programmet og endre noder med det samme navnet, siden dette

potensielt endrer noe vi ikke skulle endret. For å derfor være sikker på at vi endrer riktig referanse, kreves det en rekke komplekse og kompliserte sjekker. Det å få disse sjekkene helt fullstendige er krevende, og det er derfor mulig at det finnes scenarioer vi ikke har tatt høyde for. En grundig test-prosess ville avdekket slike feil, eller eventuelt avskrevet selve problemet.

Sjekker for endring av bindinger ved endring av navn

Som nevnt er vi nødt til å også håndtere en del semantiske aspekter samtidig som vi håndterer selve endringene av navn. Dette innebærer om en spesifisert endring fører til sideeffekter som påvirker programmet på en måte som vi har definert som ugyldig, noe som i hovedsak går på ugyldig overskriving av funksjoner og *skygging* av attributter til structer. Det vi legger i ugyldig overskriving eller skygging er at en binding til en funksjon eller et attributt endres som en følge av endring av navn. Derfor har vi definert en regel om at endring av navn som fører til enten en overskriving av funksjon eller skygging av attributt, ikke er tillatt. Vi skal videre vise eksempler på slike tilfeller, og se på hvordan vi faktisk håndterer det.

Vi vil først ta for oss dette med overskriving av funksjoner, og kan videre anta programmet under der vi har to structer `A` og `B`, der sistnevnte bruker komposisjon med `A`:

```
pttemplate T {
   type A struct { }

   func (a A) S() string {
      return "a"
   }

   type B struct {
      A
   }

   func (b B) M() string {
      return "b"
   }

   func testBSinS() string {
      b := B{A: A{}}
      return b.S()
   }
}

ptinst T with B.M => S
```

La oss se på pakkemalen `T` over, og ikke ta hensyn til instansieringen av denne. Vi kan se at vi i funksjonen `testBSinS` ville kalt på metoden `S` som er definert for `A`, noe som ville resultert i strengen "a". Dette kommer av at `B` bruker komposisjon med `A`, og siden `B` ikke har definert en metode `S` selv, vil det være metoden som er definert for `A` som blir kalt. Med andre ord er kallet "b.S()" bundet opp til `A` sin metode.

Når vi derimot tar med instansieringen av pakkemalen, kan vi se at metoden

`M` bytter navn til `S`. Det vil si at nå har `B` selv definert en metode `S`. Dette fører nå til at funksjonen `testBSinS` vil kalle på metoden til `B`, noe som ville resultert i strengen "b". Dermed har bindingen endret seg fra slik det opprinnelig var definert. Som nevnt er dette noe vi ikke ønsker å tillate, og vi kan si at metoden `S` ble ugyldig overskrevet. Dersom vi faktisk ville overskrevet denne metoden, måtte vi brukt nøkkelordet `toverride` og eksplisitt overskrevet den.

For å hindre at slike overskrivinger skal være mulig, har vi definert en sjekk for dette. Dermed vil vi for hver metode som skal endre navn, kalle på denne sjekken. Det denne gjør er å sjekke om en "forelder" allerede har definert en metode med samme navn. Med forelder mener vi en struct vi har brukt komposisjon med, og i eksempelet ville forelderen til `B` vært `A`. Denne sjekken er også rekursiv slik at dersom ikke `A` hadde definert metoden, men en forelder av `A`, så ville fortsatt sjekken fanget opp dette. Preprosessering av programmet over ville avsluttet som følge av den nevnte sjekken, og skrevet ut en feilmelding om at endringen av metode-navnet førte til en ugyldig overskriving.

Lignende problematikk slik som over gjelder også for structers felter. La oss anta følge program:

```
pttemplate T {
   type A struct {
      x string
   }

   type B struct {
      A
      y string
   }

   func testBSinX() string {
      b := B{A: A{x: "a"}, y: "b"}
      return b.x
   }
}

ptinst T with B => B(y -> x)
```

Dersom vi ser på `T` i isolasjon, kan vi se at `testBSinX` vil opprette et objekt av `B`, og siden denne structen ikke har definert attributtet `x` selv, er det attributtet som er definert for `A` som vil være det som blir returnert. Altså er "b.x" bundet opp til `A` sin `x`, og funksjonen ville dermed returnert strengen "a".

I instansieringen av `T` endrer vi navnet på feltet `y` til å nå hete "x". Uten å ta høyde for skygging av attributter vil nå `testBSinX` returnere strengen "b". Det er altså fordi `B` faktisk vil ha et felt som heter "x" etter instansiering, og bindingen i denne funksjonen endres derfor til å heller bruke det som opprinnelig var `y`.

Dette er altså et eksempel på en endring av navn som fører til skygging av attributt. Programmet over ville derfor resultert i en feilmelding om at endringen av navnet på attributtet "y" var ugyldig, og ville ført til skygging. Sjekken vi implementerte for dette ligner veldig på sjekken for ugyldig overskriving av funksjoner. Denne vil også sjekke rekursivt, slik at det ikke spiller noen rolle hvor høyt opp i "hierarkiet" et attributt med samme navn finner sted.

5.6.5 Håndtering av "addto"-setningen

Siden vi allerede har funnet hvilke pakkemaler som inneholder typer som skal utvides, kan vi nå enkelt få tak i disse AST-nodene etter at endringer av navn er håndtert. Derfor passer det bra å håndtere "addto"-setninger rett etter vi har håndtert instansieringer og endringer av navn. Dette er nok det Pakkemal-konseptet sammen med instansieringer og endring av navn som har krevet mest kode. Vi må i tillegg til å faktisk gjøre utvidelsene også her sjekke semantiske aspekter for Pakkemal-konsepter. De samme reglene vi satt for endring av navn, når det kommer til skygging av felter og overskriving av metoder, gjelder også ved utvidelse av typer. En ny funksjon eller metode kan altså ikke overskrive en eksisterende med mindre nøkkelordet `toverride` blir brukt, og et nytt felt kan heller ikke føre til skygging av et eksisterende felt.

Overskriving av metoder

Dersom en "addto"-setning inneholder metoder som skal overskrives, altså metoder som bruker nøkkelordet `toverride`, vil vi først sjekke om overskrivingen er semantisk sett gyldig. Dette innebærer å sjekke om det finnes en funksjon eller metode med samme signatur som det som er deklarert i "addto"-setningen. Hvis sjekken ikke finner en deklarasjon med lik signatur, vil preprosesseringen avsluttes og en feilmelding bli skrevet ut. Hvis den derimot finner en lik signatur, vil håndteringen av "addto"-setningen kunne fortsette.

Det neste vi så gjør er å skrive om navnet på den opprinnelige funksjonen som blir overskrevet. Det innebærer å finne deklarasjonen til denne funksjonen, og deretter endre navnet på denne til det samme som det vi gjorde om "`tsuper.Funksjon()`" til i 5.6.2.

Utvidelse av grensesnitt og structer

Etter å ha håndtert overskrivinger av metoder, kan vi nå utvide typene med det resterende som er spesifisert i "addto"-setningen. Siden vi allerede vet hvilken pakkemal som typen vi skal utvide befinner seg i, kan vi iterere listen over deklarasjoner i denne pakkemalen. Når vi så finner typen som skal utvides, må vi finne ut om det er et grensesnitt eller en struct. Dersom det er et grensesnitt, kan vi nå simpelthen sammenflette innholdet den hadde fra før med innholdet av nye metode-signaturer i "addto"-setningen.

Hvis det er en struct som skal utvides med nye felter, kan dette gjøres omtrent på samme måte som for grensesnitt. Dog må vi nå igjen sjekke at feltene som legges til ikke fører til skygging av andre felter slik som vi så i 5.6.4. For å vise hvorfor dette kan by på lignende problemer, kan vi anta programmet under:

```
pttemplate T {
   type A struct {
      s string
   }

   type B struct {
        A
   }

   func (b B) F() string {
      return b.s
   }
}

ptinst T
addto B {
   s string
}
```

La oss se på pakkemalen `T`, i kodebiten over, slik det ser ut før instansiering. Vi kan se at metoden `F` vil returnere `B` sitt attributt `s`, noe som i praksis vil si attributtet som er definert for `A`, siden `B` ikke eksplisitt har angitt et felt `s` selv. Dermed kan vi si at aksesseringen "`b.s`" er bundet opp til `A` sin `s`.

Når vi derimot tar med instansieringen og utvidelsen av `B`, kan vi se at `B` får eksplisitt et felt som er navngitt "s". Det vil føre til at dette nye feltet vil skygge `A` sin `s` i metoden `F`, og bindingen her har nå endret seg. La oss nå anta at vi oppretter et objekt av `B` og kaller på `F`:

```
b := B{A: A{s: "a"}, s: "b"}
fmt.Println(b.F())
```

Uten å ta høyde for dette med skygging av attributter, vil strengen som skrives ut være "b". Dersom bindingen i metoden ikke hadde endret seg, ville den returnert "a". Siden vi har definert en semantisk regel som sier at vi ikke skal kunne utvide en struct med nye felter som fører til skygging av andre, ville denne utvidelsen av `B` ført til terminering av preprosessering, og det ville blitt skrevet ut en feilmelding. For implementasjonen av denne sjekken kunne vi benytte den samme som vi gjorde for endring av navn, slik at vi i praksis ikke trengte å definere en ny sjekk.

For størst mulig fleksibilitet tillater vi alle slags deklarasjoner inne i "addto"-setningen. Dette gjør at vi for eksempel kan definere nye metode-deklarasjoner tilknyttet en struct både i "addto"-setningen og utenfor. Derfor vil vi som siste steg legge til alle disse andre deklarasjonene som er definert til listen over deklarasjoner i pakkemalen. Når det kommer til nye metode-deklarasjoner, er vi nødt til å sjekke om dette kan føre til ulovlig overskriving. Anta følgende program:

```
pttemplate T {
   type A struct { }

   func (a A) F() string {
         return "a"
   }

   type B struct {
         A
   }

   func kallF() string {
         b := B{A: A{}}
         return b.F()
   }
}

ptinst T
addto B {
   func (b B) F() string {
      return "b"
   }
}
```

I funksjonen `kallF` i kode-eksempelet over opprettes et objekt av `B`, og vi kaller så på funksjonen `F`. Før instansiering har ikke `B` definert denne metoden selv, men siden det brukes komposisjon med `A` vil vi ha tilgang på den. Dette gjør at "`b.F()`" er bundet opp til metoden `F` som er definert for `A`.

Etter instansiering og utvidelse av `B`, kan vi se at denne structen nå har `F` definert. Dette gjør at når vi skal kalle på `F` i `kallF` så er det denne nye metoden som blir kalt, noe som resulterer i strengen "b". Denne endringen av binding er altså et eksempel på en ugyldig overskriving. Vi har derfor en eksplisitt sjekk på dette som ville sørget for at preprosesseringen ville avsluttet, og feilmelding ville blitt skrevet ut.

5.6.6 Oppdatering av påkrevde typer

Når vi instansierer en pakkemal inne i en annen, trenger vi ikke å konkretisere påkrevde typer i pakkemalen vi instansierer. Dette er fordi disse kan videreføres i pakkemalen der instansieringen finner sted. Konkretiseringen av de påkrevde typene kan dermed utsettes til pakkemalen instansieres i det ytre skopet.

Siden vi kan la være å konkretisere påkrevde typer ved instansieringer inne i pakkemaler, bør vi før vi håndterer instansieringer av pakkemaler i det ytre skopet oppdatere hvilke påkrevde typer som enda ikke er konkretisert. Vi kan som nevnt også endre navn og utvide påkrevde typer, slik at det å oppdatere disse før vi håndterer instansieringer i det ytre skopet vil forenkle denne prosessen en hel del. Når alle påkrevde typer er oppdatert, slik at vi har kontroll på hvilke som ikke er konkretisert og hvilke krav som gjelder, kan vi håndtere påkrevde typer uten å ta høyde for hvordan endte opp som de gjorde.

For å få til dette vil vi først lete igjennom alle pakkemaler som inneholder instansieringer, og sjekke om det er noen påkrevde typer som ikke blir konkreti-

sert. Vi vil så lagre alle typene som ikke ble konkretisert, og videre sjekke om det er spesifisert noen endringer av navn for disse. Dersom dette er spesifisert, må vi også endre dem. Når de påkrevde typene skal videreføres i den nye pakkemalen, må vi ta en fysisk kopi. Dersom vi ikke hadde gjort det, ville dette potensielt ført til store sideeffekter, siden det kunne endret spesifiseringen av den påkrevde typen der den var deklarert, samt de andre stedene den ble brukt.

Dersom det er spesifisert endringer av navn, må vi også endre alle referanser av det gamle navnet i pakkemalen vi instansierer. For dette kunne vi heldigvis benytte oss av funksjonalitet vi allerede hadde utviklet for endringer av navn på vanlige typer, slik at vi ikke trengte å skrive noe særlig ny kode for å få dette til.

Koden for å håndtere sammenfletting av structer og grensesnitt fungerer også for påkrevde typer, og vi kunne dermed gjenbruke koden direkte.

5.6.7 Sammenfletting av structer og grensesnitt

For sammenfletting av flere structer til én vil funksjonaliteten for endring av navn allerede ha gjort mye av jobben som er nødvendig. Dette er fordi vi ikke trenger å gjøre noe mer med metode-deklarasjoner, siden endring av type-navn allerede har sørget for at de bruker det samme navnet. Det som derimot gjenstår er å sammenflette alle felter, og sørge for at det kun er én deklarasjon av structen. Endringene av navn har nå ført til at det er flere struct-deklarasjoner med samme navn. For å vise hvordan dette ser ut, kan vi anta følgende program før preprosessering:

```
pttemplate T1 {
   type A struct {
      a string
   }

   func (a A) GetString() string {
      return a.a
   }
}

pttemplate T2 {
   type B struct {
      b int
   }

   func (b B) GetInt() int {
      return b.b
   }
}

ptinst T1 with A => C()
ptinst T2 with B => C()
```

Vi kan nå videre anta at vi er ferdig med instansiering og de øvrige Pakkemal-konseptene som hittil er forklart i dette kapittelet. Dermed vil programmet nå se slik ut:

```
type C struct {
   a string
}

func (a C) GetString() string {
   return a.a
}

type C struct {
   b int
}

func (b C) GetInt() int {
   return b.b
}
```

Som vi ser i programmet over så er metode-deklarasjonene allerede håndtert. Dog har vi nå to deklarasjoner av `C`, slik at det som gjenstår er å samle feltene og fjerne duplikatet. Måten dette løses på er at vi bestemmer oss for én av deklarasjonene som skal bli værende, noe som blir den første deklarasjonen vi finner, og så samle opp alle feltene i denne deklarasjonen. Deklarasjonene er nå lagret i en liste, noe som gjør at vi enkelt kan lagre indeksene for duplikatene, og vi kan deretter fjerne duplikatene basert på indeksene vi lagret. I dette eksempelet ville deklarasjonen med feltet `b` blitt ansett som duplikatet, og vi ville dermed, etter at vi har samlet feltene, fjernet denne deklarasjonen. Dermed ville programmet sett slik ut:

```
type C struct {
   a string
   b int
}

func (a C) GetString() string {
   return a.a
}

func (b C) GetInt() int {
   return b.b
}
```

For sammenfletting av grensesnitt så gjøres dette i praksis på akkurat samme måte som sammenfletting av struct-deklarasjoner og felter.

5.6.8 Innsetting av konkrete typer for påkrevde typer

Det som nå gjenstår å forklare er innsettingen av konkrete typer for påkrevde typer. Dette skjer etter at alle andre Pakkemal-konsepter er tatt hånd om. Da dette skulle implementeres, vurderte vi flere ulike varianter. Blant disse var en mulig variant å faktisk ikke "fysisk" bytte ut den påkrevde typen med den konkrete. Det som menes med det er at det kunne vært mulig å utnytte det faktum at påkrevde typer og grensesnitt deler samme struktur. Siden Go håndterer typesjekkingen av grensesnitt på en strukturell måte, kunne det fungere å

representere påkrevde typer som grensesnitt. Anta denne pakkemalen:

```
pttemplate T {
   required type R {
      F(s string) string
   }

   func TestF(r R) string {
      return r.F("Test")
   }
}
```

I programmet over har vi altså en påkrevd type `R` som inneholder en metodesignatur `F`. Den formelle parameteren `r` i funksjonen `TestF` er typet som den påkrevde typen, og brukes for å kalle på funksjonen `F`. Dersom vi under preprosessering hadde endret definisjonen av `R` til å i stedet være et grensesnitt, kan vi se at funksjonen `TestF` fortsatt ville være gyldig. Koden under viser hvordan dette ville sett ut:

```
pttemplate T {
   type R interface {
      F(s string) string
   }

   func TestF(r R) string {
      return r.F("Test")
   }
}
```

Dersom vi hadde konkretisert `R` med en struct `S` som også hadde definert metoden `F`, ville dette fungert uten videre endringer. Dermed kunne vi for eksempel kalt på `TestF` med en aktuell parameter av typen `S`, uten at dette ville ført til noen problemer. Dog ville vi kunne brukt hvilken som helst type, gitt at typen hadde definert metoden `F`, siden `R` nå i praksis er et grensesnitt. Dette kan i visse scenarioer være nyttig, men akkurat for Pakkemal-mekanismen så er nok ikke dette nødvendigvis en god løsning. La oss for eksempel anta at vi i et scenario har en konkret type som har flere metoder enn det som er krevd av en påkrevd type. Dersom vi for eksempel har funksjoner som returnerer den påkrevde typen (som blir omgjort til et grensesnitt), vil vi være nødt til å caste tilbake, siden vi ellers ikke ville hatt tilgang på metoder som ikke var spesifisert for den påkrevde typen. Senere i oppgaven kommer vi litt tilbake til dette, der vi ser på muligheten for å kunne ha påkrevde typer som kan konkretiseres med grensesnitt.

Måten vi endte opp med å løse dette var ved å bytte ut alle referanser til den påkrevde typen med navnet til den konkrete typen. Dermed ville en instansiering av `T` der vi konkretiserer med structen `S`, ført til at `TestF` ville sett slik ut etter preprosessering:

```
func TestF(r S) string {
   return r.F("Test")
}
```

Vi kan dermed se at den formelle parameteren `r` nå vil være typet som structen `S`. Dette gjør at vi ikke kan kalle på funksjonen med andre structer selv om disse også har definert metoden `F`.

Siden vi allerede har sjekket at konkretiseringer av påkrevde typer er gyldige, vil den resterende jobben i praksis være å bytte ut type-navnet fra den påkrevde typen til navnet på den konkrete typen. Dette ligner veldig på endring av navn ved instansiering av pakkemaler, noe som førte til at denne koden igjen kunne gjenbrukes. Implementasjonen av dette ble derfor en nokså enkel prosess.

5.7 Den andre typesjekken

Til nå har vi parset, kjørt igjennom første typesjekk og modifisert AST-et, slik at Pakkemal-konsepter nå er tatt hånd om. Modifikasjonene på AST-et innebar i praksis å transformere et GoPT-AST til et vanlig Go-AST. Selv om vi allerede har sjekket en rekke semantiske aspekter, har vi ikke fått sjekket hele det resulterende programmet som ett. Dette må nå gjøres slik at vi er sikre på at vi ender opp med et program som er typesikkert.

Et eksempel på et program som denne sjekken ville fanget opp at var ugyldig er følgende:

```
pttemplate T1 {
   type S struct {
      a string
   }
}

pttemplate T2 {
   type O struct {
      b int
   }
}

ptinst T1
ptinst T2 with O => S()
```

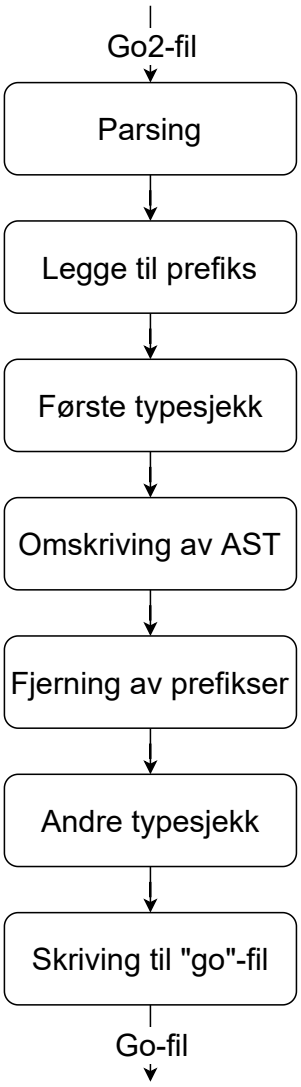
I programmet over har vi to pakkemaler `T1` og `T2`. Instansieringen av `T1` ender opp med en struct-deklarasjon med navn "S", og i instansieringen av `T2` endrer vi navnet på structen `O` til å hete "S". I 4.5 nevnte vi at for å sammenflette typer må navnene eksplisitt endres til det samme. Dette er ikke tilfellet i programmet over, og det resulterende programmet ender derfor opp med to ulike struct-deklarasjoner med navnet "S". Denne siste typesjekken vil oppdage denne feilen, altså at programmet inneholder en "redeklarasjon" av `S`, noe som vil føre til terminering og en feilmelding.

Før vi kan kjøre denne typesjekken må vi fjerne prefiksene som vi la til før den første typesjekken. Dersom vi ikke hadde fjernet disse nå, ville ikke typesjekken fanget opp problemet i programmet over. I tillegg ville det vært dumt at det resulterende programmet inneholdt nye, ukjente navn på deklarasjonene.

Dersom denne sjekken fullføres uten problemer, vil neste steg nå være å skrive til vanlig Go-fil. Implementasjonen for dette var i all hovedsak allerede implementert, og det var kun nødvendig å legge til noen få kodelinjer.

5.8 Oppsummering

Vi har nå vært igjennom implementasjonen av prototypen for GoPT. Dette har i praksis innebåret å videreutvikle preprosessoren som ble brukt for prototyping av generiske typeparametere i Go. I 5.2 viste vi hvilke aspekter preprosessoren besto av før vi gjorde endringer. Under vises en ny figur som beskriver stegene preprosessoren nå består av:



Figur 5.3: Stegene i Go sin preprosessor etter implementering av GoPT

Vi har måttet utvide parsingen slik at den nå støtter disse nye konseptene. I tillegg er det nå to runder med typesjekking, én før og én etter alle Pakkemal-konsepter er håndtert. For å kunne bruke samme typesjekk i begge tilfellene, la vi til prefikser på deklarasjoner i pakkemaler før den første typesjekken. På

steget "Omskriving av AST" har vi også gjort store utvidelser fra slik preprosessoren opprinnelig var. Når det gjelder det å skrive til vanlig Go-fil, var det kun minimale endringer og tillegg i koden som var nødvendig.

Prototypen støtter alle konseptene som ble beskrevet i kapittel 4, og vi vil videre oppsummere en del av funksjonaliteten som dermed støttes. Blant annet kan vi deklarere pakkemaler der innholdet hovedsakelig vil være deklarasjoner av structer, grensesnitt, funksjoner og andre Pakkemal-konsepter. Det å deklarere andre former for typer enn dette er noe vi kommer tilbake til. I tillegg er det slik at vi kan instansiere pakkemaler i to typer skop, enten i det ytre skopet eller i andre pakkemaler. Den samme pakkemalen kan også instansieres flere ganger, noe som ikke byr på problemer, gitt at det blir gjort nødvendig endringer av navn slik at det resulterende programmet ikke ender opp med duplikater. Structer og grensesnitt kan etter instansiering også bli videre utvidet, samt at vi kan overskrive metoder og funksjoner som allerede eksisterer i pakkemalen. Det er også mulig å sammenflette flere typer til én, og vi har i tillegg implementert støtte for generiske typeparametere i form av påkrevde typer.

Selv om konseptene som ble forklart i kapittel 4 er implementert, vil implementasjonen grunnet tidsaspektet på en slik oppgave naturlig nok ikke være feilfri. Under vil vi gjøre rede for noen aspekter som ikke fungerer optimalt eller har et forbedringspotensial:

- Koden bærer mye preg av at vi er nødt til å traversere og lete mye i AST-et. Dette er nokså naturlig siden vi håndterer det aller meste av selve støtten for Pakkemal-konsepter på AST-nivå, men ved en større refaktorering kunne vi nok hatt mer effektiv kode. Ved å for eksempel *cache* noder vi bruker mye, ville vi antagelig kunne unngått noe av traverseringen og "letingen" etter bestemte noder. I tillegg kunne vi kanskje håndtert flere aspekter samtidig i en større grad enn det vi gjør nå. Dog ville dette antagelig ført til en mer kompleks og komplisert kode, så her måtte vi eventuelt gjort en avveining mellom lesbarhet og effektivitet. Det skal også sies at ved en preprosessering slik som denne, vil det antagelig ikke være kritisk at koden er veldig effektiv, siden dette ikke påvirker for eksempel kjøretid av resulterende program. Når det kommer til effektivitet for preprosessoren, vil dette i hovedsak påvirke utviklingstiden, men siden dette er en prototype og at preprosesseringen uansett går relativt raskt, vil vi ikke anse dette som et spesielt viktig punkt.

- Som nevnt tidligere implementerer utviklerne av Go tester underveis mens de utvikler ny kode. Dette er noe vi i denne oppgaven har dratt stor nytte av, siden det har bidratt til å enkelt kunne sjekke om endringer og utvidelser vi har gjort har ført til uønskede sideeffekter. Dersom vi skulle begynt denne oppgaven på nytt, ville vi nok definert tester på samme måte som det utviklerne av Go gjør. Dette hadde gjort det enklere å raskt oppdage om endringer og tillegg i vår egen kode førte til at aspekter som vi allerede hadde implementert, ikke lenger fungerte etter nye utvidelser. Etter hvert som Pakkemal-konseptene stadig ble mer komplekse, ved å blant annet kombinere dem, kunne det være litt vanskelig å være sikker på at alt vi hadde implementert og testet tidligere fortsatt fungerte. Måten vi testet på var en mer manuell måte der vi for hvert nye konsept vi implementerte laget testfiler, men som vi måtte kjøre på en helt manuell måte. Dette tar lenger tid enn å kjøre automatiserte regresjonstester, og

det å måtte sjekke manuelt vil nok i lengden generelt sett føre til mindre testing. Konsekvensen av dette var at vi ved flere anledninger oppdaget feil sent, og det å finne årsakene til disse feilene etter at vi har implementert det er ofte vanskeligere og tar lenger tid.

- Som en følge av lite tid til grundig testing av preprosessoren, vil det være sannsynlig at det er feil og mangler ved mye nøsting av Pakkemal-konsepter. Dette er i scenarioer for eksempel der det er en rekke pakkemaler som instansierer hverandre, og det blir brukt andre Pakkemal-konsepter i tillegg. Antageligvis er dette feil og mangler av typen som går fort å rette opp, og mengden av disse er også uviss siden vi rett og slett ikke har hatt tid til å gjøre noe særlig slik testing.

- Vi har oppdaget en feil ved sammenfletting av flere structer til én dersom structene som skal sammenflettes er fra samme pakkemal. Sammenfletting fungerer helt fint dersom det er structer fra ulike pakkemaler, men dersom det er fra samme, ender vi opp med duplikater av felter. Dette er også noe som antageligvis går fort å rette opp, men på grunn av at feilen ble oppdaget veldig sent har vi ikke hatt tid til fikse dette.

Kapittel 6

Evaluering og diskusjon

I dette kapittelet vil vi diskutere valg og løsninger på ulike aspekter knyttet til prosjektet. Dette vil først innebære å se på strategi og utviklingsprosess, før vi vil gå mer i dybden på selve språket GoPT. Vi vil også gå igjennom noen større kode-eksempler, der vi i tillegg vil sammenligne GoPT med offisiell Go-kode. Til slutt vil vi se på hvordan mekanismen til syvende og sist passet språket.

6.1 Strategi og utviklingsprosess

Vi har tidligere i oppgaven forklart litt om vurderinger vi gjorde når det kom til måten vi skulle implementere disse nye konseptene i Go. Grunnen til at vi valgte å videreutvikle den nevnte preprosessoren, var hovedsakelig fordi den virket å være egnet til det vi ønsket å få til, og at dette ville føre til en rask utviklingsprosess slik at vi fikk implementert flere ulike Pakkemal-konsepter.

Dersom vi med erfaringen vi har nå skulle valgt strategi på nytt, ville vi nok antagelig valgt den samme. Når det kommer til mengden av Pakkemal-konsepter vi har rukket å implementere, er nok dette faktisk kanskje over det vi selv hadde forventet å rekke. Videreutvikling av preprosessoren muliggjorde det å nokså effektivt kunne prototype disse ulike konseptene.

Selv om vi nå kan si at det var et godt valg å velge den strategien vi gjorde, så var det også noen ulemper knyttet til videreutvikling av preprosessoren. Det var svært lite dokumentasjon av selve preprosessoren, noe som er nokså naturlig siden det kun er noen få personer som har utviklet den, og det ikke er en intensjon om å utvikle denne noe videre. Da vi støtte på problemer, var vi nødt å finne det ut av det på egen hånd. Det å stille spørsmål på for eksempel forum ga heller ikke så mye respons, siden det generelt er få som kjenner til preprosessoren, eller som i det hele tatt har interesse av videreutvikling av denne.

Til tider kunne det være en utfordring å finne en balanse mellom hvor mye kode vi skulle gjenbruke og hvor mye vi skulle skrive selv. Som en slags hovedregel har vi forsøkt å gjenbruke mest mulig kode, og vi har vært opptatt av å forsøke å unngå det å skrive kode som i praksis allerede er implementert. Eksempler på dette kan være parsing og typesjekking av vanlige konsepter i Go. Det skal dog sies at i begynnelsen av utviklingsprosessen ikke var gitt at det å for eksempel kunne gjenbruke typesjekking i pakkemal-deklarasjoner ville være mulig uten bruke alt for lang tid for å få dette til. Underveis kunne det være

fristende å skrive all denne koden selv, men sett i ettertid så var det fornuftig å gjøre gjenbruke koden.

Et av grepene vi gjorde for å kunne gjenbruke typesjekking var som nevnt tidligere å legge til et prefiks på alle deklarasjoner i pakkemaler. Fra begynnelsen av implementasjonen fjernet vi ikke disse før etter modifikasjonene på AST-et var ferdig. Dette virket fornuftig i starten siden det var nokså enkelt, men sett i ettertid burde vi antageligvis gjort dette på en annen måte, eller i alle fall fjernet disse prefiksene før vi begynte å gjøre modifikasjoner på AST-et. Siden deklarasjonene hadde et prefiks mens vi modifiserte AST-et, måtte vi også ta høyde for dette her, noe som resulterte i en god del flere sjekker. Etter hvert som konseptene ble mer komplekse, så kompliserte dette en del, men siden resten av koden allerede antok disse prefiksene ville det også blitt en nokså stor jobb å gjøre om på dette.

Det å finne en balanse mellom hvor mye tid som skulle gå til testing, og hvor mye tid som skulle brukes på implementasjon av nye konsepter, var også en vanskelig avveining. Antageligvis ville det vært fordelaktig å bruke enda mer tid på testing, og som vi har vært inne på tidligere laget automatiserte tester. Kanskje kunne vi også fulgt prinsipper fra *testdrevet utvikling* [6], som går på å skrive tester før vi implementerer selve koden. Ved flere anledninger fant vi feil som egentlig burde vært håndtert på et mye tidligere tidspunkt. Det å rette opp en feil i implementasjonen av et nytt konsept er mye enklere å gjøre på tidspunktet hvor vi implementerer det, og en konsekvens av dette kunne dermed vært at implementasjonen hadde vært enda litt mer effektiv.

Helt i starten av implementeringen var det også litt utfordrende siden preprosessoren ikke var helt ferdig, og som en følge av dette inneholdt en rekke feil. Dog tok det ikke veldig lang tid fra vi startet til de aller fleste feilene var rettet opp. Siden preprosessoren kun var en prototype for Google, var det likevel noen feil som aldri ble rettet opp, og som en følge av dette førte til litt uklarheter om det vi forsøkte å få til ikke var mulig i Go, eller om det var en feil i preprosessoren. Et eksempel på dette kommer vi litt tilbake til i 6.3.2.

6.2 GoPT

Fram til nå har vi sett mest på konsepter som faktisk er implementert og som fungerer i preprosessoren. I løpet av prosjektet har vi kommet på andre konsepter eller videre funksjonalitet på eksisterende konsepter, men som vi ikke har hatt tid til å se nærmere på. Dette innebærer enten at vi ikke har hatt tid til selve implementeringen, eller at vi ikke har fått undersøkt konseptene nok til å avgjøre om de vil ha en nytteverdi, eventuelt at introduksjonen av dette vil føre til komplikasjoner.

6.2.1 Instansieringer av pakkemaler i andre skop

Som nevnt støtter GoPT instansiering av pakkemaler i det ytre skopet og i andre pakkemaler. Dog kunne det kanskje vært nyttig å utvide dette slik at GoPT også hadde støttet instansieringer i andre typer skop. Eksempler på slike skop kunne vært inne i funksjoner eller løkker. Under følger et kode-eksempel som viser bruk av dette der pakkemalen `T` instansieres i funksjonen `F` og i en løkke inne i funksjonen.

```
pttemplate T {
   type S struct {
      ...
   }
}

func F() {
   ptinst T
   ...
   for {
      ptinst T with S => P()
      ...
   }
}
```

I eksempelet over hadde vi hatt tilgang på structen `S` innen hele funksjonen `F`. Denne deklarasjonen hadde dog vært begrenset slik at vi ikke hadde hatt tilgang på den utenfor. Det samme gjelder for structen `P` som hadde vært tilgjengelig innenfor løkken, men for eksempel ikke i resten av funksjonen `F`.

Go støtter deklarasjoner av for eksempel structer inne i funksjoner og løkker, og derfor kunne det kanskje vært naturlig å også tillate instansieringer av pakkemaler i slike skop. Dog skal det sies at det i tidligere implementasjoner av Pakkemal-mekanismen ikke har vært ansett å være særlig hensiktsmessig å tillate dette. Bruksområdet til Pakkemal-mekanismen er i hovedsak når det er snakk om en stor mengde deklarasjoner, og i scenarioer der disse kan gjenbrukes. Derfor vil det nok i praksis være nokså sjeldent at det å for eksempel instansiere pakkemaler i løkker vil være spesielt nyttig.

6.2.2 Andre typer deklarasjoner i pakkemaler

Til nå har vi sagt at det som i hovedsak kan deklareres inne i pakkemaler er grensesnitt, structer, funksjoner og andre Pakkemal-konsepter. Senere i oppgaven skal vi likevel se at det er mulig å deklarere en konstant og bruke denne som andre deklarasjoner i pakkemaler. Dette er noe vi ikke eksplisitt har implementert støtte for, og vi kan dermed ikke for eksempel endre navn på denne.

Dersom vi hadde hatt mer tid i dette prosjektet, ville vi nok sett på mulighetene for hvilke andre typer deklarasjoner vi kunne brukt i pakkemaler, og hvilke Pakkemal-konsepter vi videre kunne brukt på disse. Det mest nærliggende ville kanskje være å eksplisitt støtte variabel- og konstant-deklarasjoner av basistyper. Siden det å deklarere dette allerede fungerer, ville jobben i hovedsak bestått av å utforske hvilke Pakkemal-konsepter vi kunne brukt på dem. Antageligvis ville det å tillate å endre navn på disse deklarasjonene vært nyttig. For dette måtte vi eventuelt funnet en syntaks som ikke ville ført til konflikter med det som allerede er tillatt. Det å kunne bruke andre Pakkemal-konsepter på disse deklarasjonene som "addto" og sammenfletting ville trolig ikke gitt så mye mening. Derimot kunne det å overskrive med bruk av `toverride` kanskje vært nyttig.

På samme måte som variabel- og konstant-deklarasjoner av basis-typer i pakkemaler, tillater vi også det samme for sammensatte typer. Det er naturligvis allerede full støtte for bruk av structer og grensesnitt, men andre sammensatte

typer som for eksempel lister og "maps" er begrenset. Vi kan deklarere slike typer og bruke "vanlig" Go-konsepter på disse, men vi har ikke tatt hensyn til bruk av Pakkemal-konsepter på dem. Antageligvis ville det vært nyttig å tillate dette, men det å for eksempel utvide eller sammenflette slike typer er noe vi måtte undersøkt nærmere.

Det er sannsynlig at det å tillate variabel-deklarasjoner inne i pakkemal-deklarasjoner også fører ved seg en del potensielle problemer. Dersom disse deklarasjonene for eksempel hadde benyttet seg av type-inferens, er det mulig dette ville ført til komplikasjoner, særlig dersom vi også hadde tillatt å overskrive dem. I slike scenarioer burde det nok blant annet sjekkes at den nye verdien evalueres til samme type som den opprinnelige.

Datautvekslingsformatet *JSON* er noe vi ikke har snakket om hittil i denne oppgaven. Dette er et format som blir mye brukt i nettverksprogrammering, og som det er god støtte for i Go. For eksempel har språket eksplisitt syntaks for "JSON-structer", noe som gjør koding og dekoding av JSON smidig. Det som kjennetegner JSON er at det er nokså dynamisk og fleksibelt. Et av hovedmålene for Pakkemal-mekanismen er jo at den skal være fleksibel, så kombinasjonen av dette kunne muligens fungert godt sammen.

6.2.3 Påkrevde typer

Påkrevde typer kan som kjent kun konkretiseres med structer, men det å utvide dette med å kunne konkretisere med andre typer ville kanskje gjort mekanismen enda litt mer fleksibel. Dersom vi hadde hatt mer tid, ville vi nok først og fremst sett på muligheten for å kunne konkretisere med grensesnitt. Fra erfaringene vi har fått i denne oppgaven ville det antageligvis i så fall vært hensiktsmessig å eksplisitt skille mellom påkrevde grensesnitt og påkrevde structer. Dette er hovedsakelig fordi vi da mest sannsynlig ville tillatt mer spesifikke struct- og grensesnitt-krav, og for å kunne ha kontroll ved andre Pakkemal-konsepter som utvidelse og sammenfletting, ville det nok være lurt å eksplisitt skille de to.

I løpet av prosjektet har vi også diskutert om vi burde kunne angi felter i deklarasjonen av en påkrevd type som en struct må ha definert for å kunne konkretisere den påkrevde typen. Dette kan være nyttig dersom vi for eksempel kun er interessert i spesifikk data som en type må inneholde. La oss si vi vil kreve at en struct for eksempel må ha en id, noe som typisk er uttrykt ved et felt i en struct. Ved å ha følgende påkrevd type, kunne vi dermed fått uttrykt dette:

```
required type R {
   ID string
}
```

Kravet i den påkrevde typen over ville blitt innfridd med alle structer som dermed inneholder dette feltet. Dersom vi skulle hatt et tilsvarende krav i vår versjon, måtte vi uttrykt dette med en metode, og den påkrevde typen hadde dermed sett slik ut:

```
required type R {
   func ID() string
}
```

De to påkrevde typene over er nokså like, og bruker omtrent like mye tekst. Likevel kan det tenkes at i scenarioer der vi trenger mange ulike krav til data, vil det kunne være unødvendig å måtte definere mange metoder som kun returnerer data fra structen. Slike metoder vil typisk kun returnere et av feltene i structen, og dette kunne vi like så godt aksessert direkte i stedet for å få tak i dataen via metoder.

Dersom påkrevde typer både skulle støttet konkretisering av grensesnitt og støtte for spesifisering av felter i den påkrevde typen, ville det nok vært en stor fordel å skille mellom påkrevde grensesnitt og påkrevde structer. Hvis vi ikke hadde skilt mellom disse, ville det antageligvis ført til problemer dersom en påkrevd type hadde inneholdt felter og vi hadde konkretisert med et grensesnitt.

For en del mekanismer som støtter generiske typeparametere kan man konkretisere med vanlige basis-typer. Dette er også mulig med Go sitt forslag, der dette uttrykkes ved å liste opp typer i et grensesnitt som den kan konkretiseres til. Under er et eksempel der skranken er at typene må kunne sorteres:

```
type Ordered interface {
   type int, int8, int16, int32, int64,
      uint, uint8, uint16, uint32, uint64, uintptr,
      float32, float64,
      string
}
```

Dette er dog ikke et gyldig grensesnitt i den gjeldende offisielle versjonen av Go, men når generiske typeparametere blir implementert i standardkompilatoren, vil dette antageligvis være mulig. Noe lignende kunne vi kanskje gjort for påkrevde typer der vi i deklarasjons-kroppen kunne spesifisert typer den kunne blitt konkretisert til. For slike påkrevde typer ville det antageligvis igjen vært fornuftig å skille dette ut til "påkrevde basis-typer". Det kan tenkes at konsepter som utvidelse og sammenfletting også ville vært nyttig for dette. Antageligvis burde vi begrenset endring av navn til å kun være tilgjengelig for navnet på den påkrevde typen, og ikke nødvendigvis innholdet til denne typen. Det å skulle endre navn på basis-typene ville antagelig ført til veldig store sideeffekter.

I denne versjonen av GoPT har vi ikke implementert all støtte for utvidelse og tilpassing av påkrevde typer i det ytre skopet som det som er mulig inne i pakkemaler. Siden vi er nødt til å konkretisere pakkemaler i det ytre skopet, vil ikke det å utvide eller sammenflette påkrevde typer gi noe spesiell verdi. La oss anta følgende program:

```
pttemplate T {
   required type R {
      A()
   }
}

type S struct { }
func (s S) A() { }
func (s S) B() { }
```

I programmet over kunne vi i prinsippet utvidet den påkrevde typen til å også kreve at den må ha metoden `B`, og fortsatt konkretisert med structen `S`.

Dersom det i en annen pakkemal var en påkrevd type med metoden `B` kunne vi også sammenflettet de to, og gjort tilsvarende. Dog må jo disse konkretiseres samtidig som de utvides, slik at det å gjøre dette har ingen praktisk betydning.

La oss anta at metode-deklarasjonen `A` ikke eksisterer over. Det som kunne vært nyttig i dette scenarioet var om vi kunne endret navn på `R` sin `A` til "B", slik at konkretiseringen av `R` med `S` på denne måten ville blitt gyldig. Siden både instansieringen og konkretiseringen skjer i det ytre skopet, må vi både endre navn på `A` og konkretisere `R` i samme "ptinst"-setning. Dette ville eventuelt sett slik ut:

```
pttemplate T {
   required type R {
      A()
   }
}

type S struct { }
func (s S) B() { }

ptinst T with R => R(A -> B), R <= S
```

I scenarioer der vi både er nødt til å håndtere konkretisering og endring av navn samtidig, er det et problem med rekkefølgen dette skjer. Det er antagelig ikke vanskelig å endre implementasjonen slik at dette fungerer, men det fører altså til problemer slik det er nå. For øvrig er det kun tilfeller slik som beskrives over som vil føre til problemer, og endring av navn på påkrevde typer og dens krav inne i pakkemaler fungerer slik som vist tidligere i oppgaven.

6.2.4 Importering av pakker

Denne versjonen av GoPT støtter bruk av importerte pakker inne i pakkemaler. Måten vi gjør dette på er ved å importere pakkene i det ytre skopet, slik som alle andre pakker. Disse pakkene kan så brukes fritt inne i pakkemalen på helt vanlig måte. Under følger et eksempel:

```
import "fmt"

pttemplate T {
   func PrintHelloWorld() {
      fmt.Println("Hello World")
   }
}
```

I eksempelet over kan vi se at funksjonen inne i pakkemalen `T` bruker pakken `fmt` for å skrive ut "Hello World". Dette er mulig, siden vi i det ytre skopet har importert pakken som brukes.

I denne oppgaven har vi ikke hatt tid til å grundig undersøke hvordan vi burde løse dette på best mulig måte. Slik det fungerer i eksempelet over vil pakkemalen nå avhenge av ytre faktorer for at den skal være gyldig. Dersom vi ikke hadde importert `fmt` i det ytre skopet, ville ikke den resulterende koden etter instansiering vært gyldig. Vi har derfor lagt inn en sjekk som avslutter preprosessering og skriver ut en feilmelding dersom dette er tilfellet. Dette vil

dog si at vi ikke lenger kun kan forholde oss til pakkemalen som en isolert enhet, men må passe på at pakker den bruker faktisk blir importert i skopet det skal instansieres i.

En annen mulig variant vi så vidt har diskutert, er om de importerte pakkene som en pakkemal bruker burde spesifiseres inne i selve pakkemal-kroppen. Eksempelet over ville i så måte sett slik ut:

```
pttemplate T {
   import "fmt"

   func PrintHelloWorld() {
      fmt.Println("Hello World")
   }
}
```

Dersom vi hadde gjort det på denne måten, ville vi kunne avgjøre om en pakkemal er semantisk gyldig uten å ta hensyn til ytre faktorer. Innenfor pakkemal-kroppen ville vi hatt all nødvendig informasjon for å avgjøre dette. Hvis vi skulle løst det slik, måtte vi passet på at importeringene i det ytre skopet ikke endte med duplikater av pakker, siden dette ikke er tillatt i språket.

I denne oppgaven har vi heller ikke sett spesielt mye på hvordan pakkemaler vil kunne distribueres. Det mest sannsynlige ville kanskje vært å definere pakkemaler i vanlige Go-pakker, slik at vi på denne måten kunne utnyttet funksjonaliteten som tillater å enkelt dele Go-kode globalt. Dersom dette hadde vært tilfellet, kunne vi hatt samme strategi som i nåværende versjon av GoPT, altså ved å importere nødvendige pakker i det ytre skopet. På denne måten ville vi nemlig vært sikker på at skopet som pakkemalen instansieres i har importert alle nødvendige pakker. Dette er fordi pakken der pakkemalen er deklarert må importeres selv, slik at dette fører til at alle pakker som blir importert i pakkemal-pakken også blir importert der pakkemalen vil bli brukt og instansiert.

6.3 Større eksempler og sammenligning med generiske typeparametere

Til nå har vi kun sett på små eksempler med bruk av GoPT, og vi vil nå se på litt større eksempler der vi viser hvordan denne mekanismen kan brukes til litt mer praktiske formål. I teksten vil vi abstrahere bort detaljer knyttet til selve implementasjonen, men komplette kjørbare filer av eksemplene er å finne på [32].

6.3.1 Avspiller-eksempel

La oss anta at vi ønsker å utvikle to ulike avspillere, henholdsvis for avspilling av sanger og bildefiler. For begge disse scenarioene ønsker vi å legge til og fjerne elementer i en kø. Derfor har vi i kode-eksempelet under definert en pakkemal som implementerer en lenket liste. For strukturen av lenkede lister i eksempelet under tok vi utgangspunkt i kode-eksempelet for liste-transformasjon i [26].

```
pttemplate LinkedListTemplate {
   required type R {
      // any
   }

   type element struct {
      next *element
      val  R
   }

   type LinkedList struct {
      head, tail *element
   }

   func (lst *LinkedList) PushToTail(v R) {
      ...
   }

   func (lst *LinkedList) PushToHead(v R) {
      ...
   }

   func (lst *LinkedList) RemoveAtTail() (R, bool) {
      ...
   }

   func (lst *LinkedList) RemoveAtHead() (R, bool) {
      ...
   }
}
```

Pakkemalen `LinkedListTemplate` har funksjonalitet som tillater å legge til og fjerne elementer på starten og slutten i en lenket liste. Vi benytter oss av en påkrevd type som ikke har noen krav, slik at vi kan lagre alle slags structer.

Denne pakkemalen er veldig generell og kan brukes til en rekke formål. Det vi ønsket var å utvikle et program som kunne avspille sanger og bilder, og vi definerer derfor en ny pakkemal som benytter seg av `LinkedListTemplate`, men tilpasser og utvider funksjonaliteten ytterligere. Anta derfor pakkemalen `PlayerTemplate`:

```
pttemplate PlayerTemplate {
   ptinst LinkedListTemplate with
      LinkedList.RemoveAtHead => SkipElement,
      LinkedList.PushToTail => QueueElement,
      LinkedList => Player()

   addto required type R {
      PlayElement()
   }
```

```
   addto Player {
      func (lst *Player) PlayFirstElement() {
         lst.head.val.PlayElement()
      }
   }
}
```

I instansieringen av `LinkedListTemplate` endrer vi en rekke navn, slik at disse blir mer presise til avspiller-formålet. Vi legger også til et krav i den påkrevde typen om at structer som konkretiserer denne må ha en metode som kan spille av elementer, og siden denne ikke konkretiseres vil den bli videreført i denne pakkemalen. I structen `Player` (som opprinnelig het "LinkedList") har vi utvidet med en metode som spiller av første element. Vi kan også se at vi i denne metoden bruker det nye kravet til den påkrevde typen `R`, nemlig metoden `PlayElement`.

For å nå gjøre det mulig å spille av sanger, må vi instansiere `PlayerTemplate` og konkretisere den påkrevde typen med en struct som har støtte for å avspille sanger. I eksempelet nedenfor simulerer vi det å spille av sanger ved å skrive ut tekst:

```
type AudioPlayerImpl struct {
   song string
}

func (a AudioPlayerImpl) PlayElement() {
   fmt.Println("Playing the song ", a.song)
}

ptinst PlayerTemplate with
   R <= AudioPlayerImpl,
   Player.PlayFirstElement => PlaySong,
   Player.SkipElement => SkipSong,
   Player.QueueElement => QueueSong
```

Structen `AudioPlayerImpl` har definert metoden `PlayElement`, og innfrir dermed kravene til den påkrevde typen `R`. Vi kan også se at vi i instansieringen av `PlayerTemplate` tilpasser pakkemalen ytterligere, slik at deklarasjons-navnene blir enda litt mer presise.

Nå som pakkemalen er instansiert kan vi benytte oss av deklarasjonene og funksjonaliteten til pakkemalen. Under følger det kode som viser bruk av den, der vi legger til, spiller av og fjerner sanger:

```
audioplayer := Player{}

audioplayer.QueueSong(AudioPlayerImpl{"Dreamsicle"})
audioplayer.QueueSong(AudioPlayerImpl{"Last of My Kind"})
audioplayer.QueueSong(AudioPlayerImpl{"If We Were Vampires"})

audioplayer.PlaySong()
song, _ := audioplayer.SkipSong()
fmt.Println("Skipping song ", song)
audioplayer.PlaySong()
```

I eksempelet over kan vi nå se at vi bruker de nye navnene på deklarasjoner og metoder. Det gjør at vi ikke trenger å ta særlig hensyn til hvordan den underliggende strukturen er definert, og at vi dermed til en større grad kun trenger å ta hensyn til funksjonaliteten som tilbys. Dersom for eksempel ikke dette med endring av navn hadde vært støttet, måtte vi kalt på "PushToTail" i stedet for "QueueSong". Eventuelt måtte vi kanskje deklarert en ny struktur utelukkende for avspilling av sanger. Sammenligninger mellom Go og GoPT er noe vi skal se nærmere på i 6.3.2.

Dersom vi nå ønsket å utvikle en bildefremviser, ville vi kunne gjenbrukt `PlayerTemplate`. Denne pakkemalen kunne igjen vært instansiert, og tilpasset for dette formålet. For å konkretisere den påkrevde typen, ville vi vært nødt til å definere en struct med metoden `PlayElement` som ville håndtert de spesifikke aspektene ved å vise bilder. Structen og metoden er for øvrig de eneste deklarasjonene vi måtte definert, siden vi kunne gjenbrukt deklarasjonene og funksjonaliteten til `PlayerTemplate`.

6.3.2 Graf-eksempel

Et mye brukt eksempel for å vise nytten av Pakkemal-mekanismen omhandler bruk av en graf. I det neste eksempelet skal vi også benytte oss av en graf-struktur, der vi ender opp med en modellering av et toglinje-nettverk. Vi vil for dette eksempelet lage to utgaver, én med bruk av Pakkemal-mekansimen og én med bruk av vanlig Go (inkludert støtte for generiske typeparametere), og sammenligne disse to utgavene.

Graf-struktur

La oss først ta for oss en pakkemal som inneholder struktur for å uttrykke grafer der vi kan ha vekter i mellom nodene. Denne pakkemalen er egentlig over 150 kodelinjer, men vi vil i denne teksten strippe den ned til kun det nødvendige for dette eksempelet. Det vi ønsker med denne pakkemalen er å kunne legge til noder og kanter, der kantene også forteller noe om kostnaden mellom to noder. Noder har vi uttrykt med en egen struct som inneholder et felt med selve verdien. Dette feltet er typet som en påkrevd type, og har kun et krav om å returnere tekst om noden. Kant-structen inneholder to felter av typen `Node`, samt et felt som forteller noe om kostnaden mellom de to. Denne kostnaden er også en påkrevd type, der kravet er at den konkrete typen må inneholde metoden `GetCost`. Vi har også definert en struct `Graph` som lagrer alle noder og kanter, og som også håndterer strukturen mellom disse. Under følger en forenklet versjon av pakkemalen:

```
pttemplate GraphTemplate {
   const Infinity = int(^uint(0) >> 1)

   required type NodeItem {
      String() string
   }

   required type EdgeCost {
      GetCost() int
   }

   type Node struct {
      item NodeItem
   }
   ...
   type Edge struct {
      from *Node
      to *Node
      edgeCost EdgeCost
   }

   type Graph struct {
      nodes []*Node
      edges map[Node][]*Edge
   }

   ...
}
```

Structen `Graph` har i tillegg til koden som vises over en rekke metoder som sørger for lagring av noder og kanter, i tillegg til å kunne finne korteste vei mellom ulike noder (både med og uten hensyn til kostnad).

Vi har tidligere nevnt at det er mulig å deklarere konstanter i pakkemaler, og dette ser vi også et eksempel på i pakkemalen over. Konstanten `Infinity` er et tall som brukes i metodene for å finne korteste vei. Denne kan for øvrig brukes på lik måte som konstanter ellers i Go, og vil også bli "pakket ut" når pakkemalen instansieres.

Dersom vi skulle utviklet en lignende struktur som `GraphTemplate` med bruk av vanlig Go, ville det trolig være naturlig å bruke grensesnitt i stedet for påkrevde typer. Dette hadde dog ført til at for eksempel `item` i `Node` kunne vært av hvilken som helst struct så lenge den hadde definert metoden `String`.

Siden forslaget om generiske typeparametere i Go nå er godkjent, og dermed under utvikling i standardkompilatoren, har vi forsøkt å utvikle en alternativ variant med bruk av den nye typeparameter-støtten der vi ønsker å få til det samme som med pakkemaler. Dette språket, Go med støtte for generiske typeparametere, kaller vi videre for *Go2*. Koden under viser hvordan `GraphTemplate` kunne vært implementert med Go2:

```
const Infinity = int(^uint(0) >> 1)

type NodeItem interface {
   comparable // due to key in map and use of "=="/"!="
   String() string
}

type EdgeCost interface {
   GetCost() int
}

type Node[T NodeItem] struct {
    item T
}

type Edge[T NodeItem, C EdgeCost] struct {
   from *Node[T]
   to *Node[T]
   edgeCost C
}

type Graph[T NodeItem, C EdgeCost] struct {
   nodes []*Node[T]
   edges map[Node[T]][]*Edge[T, C]
}

...
```

Vi kan se over at de påkrevde typene nå er byttet ut med grensesnitt som representerer skrankene til de generiske typeparameterne. Det at `NodeItem` har definert at den må være sammenlignbar er for øvrig for å kunne bruke typen som en nøkkel i "maps" og for sammenligning. Dette slipper vi med GoPT, siden vi uansett vil konkretisere med structer, noe som er en sammenlignbar type. Ellers kan vi se at koden er nokså lik, men vi kan også legge merke til at vi er nødt til å gjenta disse generiske typene ganske mye mer enn det vi er nødt til med påkrevde typer. Dette ser vi også spesielt ved metoder og når vi skal opprette instanser av structer, noe vi skal forsøke å tallfeste senere.

Geografisk nettverk

Videre har vi utviklet en ny pakkemal `GeoNetworkTemplate`, der intensjonen var å tilby funksjonalitet og struktur for modellering av geografiske nettverk. Dette har vi vært inne på tidligere, og kunne for eksempel blitt brukt for å representere veinett eller flyrutenett, som ble beskrevet i innledningen og kapittel 2. I praksis er dette en utvidelse av `GraphTemplate`, der grafen nå er rettet mer mot geografiske noder, og der kantene også vil fortelle noe om fysisk avstand. Anta derfor følgende pakkemal:

```
pttemplate GeoNetworkTemplate {
   required type MeansOfTransport {
      String() string
      TimeBetweenTwoPlaces(road *Road) int
   }

   ptinst GraphTemplate with
      EdgeCost <= TravelTime,
      Graph.AddNode => AddPlace,
      Graph.AddEdge => AddRoad,
      Graph => GeoNetwork(nodes -> places),
      Edge => Road(edgeCost -> travelTime),
      Node => Node(item -> location),
      NodeItem => GeoLocation(),
      Graph.Dijkstra => CalculateShortestRoutes

   addto struct Road {
      meansOfTransport *MeansOfTransport
      km int
   }

   addto required type GeoLocation {
      GetCoordinates() (float32, float32)
   }

   type TravelTime struct {
      cost int
   }

   func (t TravelTime) GetCost() int {
      return t.cost
   }
}
```

I pakkemalen over har vi definert en ny påkrevd type som representerer transportmiddel, der den konkretiserte typen både må kunne returnere informasjon om den selv, samt kalkulere hvor lang tid den bruker mellom to plasser. I tillegg instansieres `GraphTemplate`, og vi spesifiserer en rekke endringer av navn, slik at dette forhåpentligvis kan abstrahere bort detaljer knyttet til implementasjon. Eksempler på dette kan være at vi ikke lenger refererer til "noder" og "kanter", men heller bruker begreper som "vei" og "sted". For øvrig kan vi se at vi refererer til metoden `Dijkstra`, som er en metode som ikke er med i kodeutdraget for `GraphTemplate` over, men som er del av den komplette koden. Vi utvider også `Road` (som opprinnelig het "Edge") slik at denne nå inneholder informasjon om både fysisk strekning og hva slags transportmiddel som kan benyttes på denne strekningen.

Den påkrevde typen `NodeItem` blir i instansieringen endret til å nå hete "GeoLocation". Videre i koden utvider vi denne påkrevde typen slik at den nå krever en metode som returnerer koordinater. I instansieringen av `GraphTemplate` konkretiserer vi også `EdgeCost` med structen `TravelTime` som er definert i samme pakkemal. Her ser vi for øvrig et eksempel på et scenario der det ville vært gunstig å kunne kreve felter i kroppen til påkrevde typer, slik som foreslått i

6.2.3. Dersom vi i den påkrevde typen `EdgeCost` kunne krevd et felt `cost`, så ville det holdt med struct-deklarasjonen for `TravelTime`, og vi ville sluppet å definere metoden `GetCost`.

Go2-implementasjonen av geografisk nettverk

Pakkemalen `GeoNetworkTemplate` innebar i stor grad å tilpasse den allerede eksisterende pakkemalen `GraphTemplate`. I neste kode-eksempel skal vi se på hvordan noe lignende ville sett ut med bruk av Go2. Siden koden i `GeoNetworkTemplate` hovedsakelig bruker Pakkemal-konsepter, er det naturlig at de to ulike implementasjonene vil divergere en hel del.

For å legge til `MeansOfTransport` så innebærer dette å definere et nytt grensesnitt som er omtrent likt som innholdet til den påkrevde typen i GoPT-eksempelet:

```
type MeansOfTransport[T GeoLocation, C EdgeCost] interface {
   String() string
   TimeBetweenTwoPlaces(edge *Road[T, C]) int
}
```

Det å få til noe lignende som "addto"-setningen kan vi til en viss grad få til ved å bruke komposisjon. Dette krever dog at vi, i tillegg til å utvide typen, også endrer navn. Det er tilfellet når vi utvider `GeoLocation`, siden denne i instansieringen byttet fra "NodeItem" til "GeoLocation". På grunn av dette kan vi bruke komposisjon og "utvide" slik:

```
type GeoLocation interface {
   NodeItem
   GetCoordinates() (float32, float32)
}
```

De opprinnelige kravene ("comparable" og "String") vil dermed bli overført til det nye grensesnittet `GeoLocation`. Måten vi med bruk av Go2 får til noe lignende som "addto"-setningen er semantisk nokså annerledes fra GoPT, og byr som sagt på problemer dersom vi ikke hadde gitt et nytt navn. I GoPT-eksempelet finner vi den faktiske typen og utvider denne, mens vi her bare definerer et helt nytt grensesnitt.

I pakkemalen `GeoNetworkTemplate` utvider vi også `Road` til å inneholde to nye felter. Siden denne structen har byttet navn, kan vi også her bruke komposisjon for å få til noe lignende som "addto". Dog fungerer det ikke å bruke komposisjon med en struct som benytter seg av generiske typeparametere. Dette vanskeliggjør det litt, siden vi dermed ikke vil kunne aksessere attributter definert for `Edge` direkte, og vi må i stedet benytte `Edge` som et felt. Vi har for øvrig undersøkt, ved å stille spørsmål på diverse forum, om dette er et design-grep for Go2 eller om det kun er en feil ved denne prototypen. Svarende vi fikk tydet på at det var en feil som vil bli rettet opp i implementeringen i standardkompilatoren, men dette er fremdeles litt usikkert. Structen `Road` vil i Go2-eksempelet se slik ut:

```
type Road[T GeoLocation, C EdgeCost] struct {
   edge *Edge[T, C]
   meansOfTransport MeansOfTransport[T, C]
   km int
}
```

Endringer av navn, som vi benytter oss flittig av ved instansiering av `GraphTemplate`, er det er veldig liten støtte for med Go2. Det eneste verktøyet som til en viss grad kan hjelpe oss med dette er "aliasing", noe vi skal se på senere. Ellers består det meste av koden av å definere nye typer som bruker den opprinnelige, eller for metoder at vi definerer en ny med det nye navnet, og at vi deretter kaller på den opprinnelige metoden. I instansieringen av `GraphTemplate` endret vi i GoPT-eksempelet `Road` (som opprinnelig het "Edge") sin "edgeCost" til å i stedet hete "travelTime". Med bruk av Go2 er det ikke mulig å endre navn på structers felter, noe som gjør at vi i Go2-eksempelet ikke får endret navnet "edgeCost". Dersom vi i Go2-eksempelet skulle brukt "travelTime" i stedet for "edgeCost", måtte vi antagelig deklarert en ny struct for `Road` som ikke gjenbrukte `Edge` slik som i dette eksempelet. Vi valgte derfor å beholde det opprinnelige felt-navnet, og heller gjenbruke koden til `Edge`.

Det samme problemet støter vi på ved endring av navn på `Graph`, fordi feltet `nodes` egentlig skal bytte navn til "places". Siden de to mulighetene vi da har er å enten la feltet fortsatt hete "nodes", eller deklarere en helt ny struct uten noen form for gjenbruk, valgte vi å la navnet "nodes" bli værende. Under viser vi hvordan vi får til noe tilsvarende som endring av navn på `Graph`, og "endring" av navn på metoden `AddEdge`:

```
type GeoNetwork[T GeoLocation, C EdgeCost] struct {
   graph *Graph[T, C]
}

func (g *GeoNetwork[T, C]) AddRoad(fromEdge, toEdge *Road[T, C]) {
   g.graph.AddEdge(fromEdge.edge, toEdge.edge)
}
```

I kode-eksempelet over kan vi altså se at vi i praksis kun deklarerer en ny type og en ny metode som benytter seg av det som opprinnelig var definert.

Instansiering og bruk av geografisk nettverk

Vi skal videre benytte oss av pakkemalen `GeoNetworkTemplate`, for å modellere et toglinje-nettverk. Pakkemalen inneholder nå to påkrevde typer, henholdsvis `MeansOfTransport` og `GeoLocation`. Derfor har vi definert to nye structer, `Train` og `Place`, som tilfredsstiller kravene til de påkrevde typene og som vi dermed bruker til konkretiseringene. Vi gjør også noen videre tilpasninger av navn, og hele instansieringen ser slik ut:

```
ptinst GeoNetworkTemplate with
   MeansOfTransport <= Train,
   GeoLocation <= Place,
   GeoNetwork => RailMap(),
   Road => RailLine(),
   Graph.AddRoad => AddRailLine,
```

```
   Graph.AddPlace => AddCity,
   Node => TrainStation()
```

Vi kan nå legge til byer og toglinjer, og for eksempel finne korteste rute fra en by til en annen. Under følger litt test-kode som benytter seg av den instansierte pakkemalen:

```
var railMap RailMap
aberdeen := TrainStation{location: Place { name: "Aberdeen", latitude:
    57.14, longitude: -2.10 } }
...
railMap.AddCity(&aberdeen)

aberdeenToBrussel := &RailLine{from: &aberdeen, to: &brussel,
    meansOfTransport: &Train{"train"}, km: 1220.0}
aberdeenToBrussel.travelTime = TravelTime{cost:
   aberdeenToBrussel
      .meansOfTransport
         .TimeBetweenTwoPlaces(aberdeenToBrussel)}
...
railMap.AddRailLine(aberdeenToBrussel, brusselToAberdeen)
...
path, _ := railMap.ShortestPath(aberdeen, elche)
...
fmt.Println("\n", railMap.CalculateShortestRoutes(&aberdeen))
```

For Go2-eksempelet vil "instansiering av GeoNetworkTemplate"-steget i likhet med GoPT-eksempelet innebære å definere structer for konkretisering av generiske typeparametere. På dette steget kan vi faktisk bruke aliasing for å "endre" navn på det vi skal aksessere. Under vises det hvordan dette gjøres:

```
type RailLine = Road
type RailMap = GeoNetwork[Place, TravelTime]
type TrainStation = Node
var AddCity = (*RailMap).AddPlace
var AddRailLine = (*RailMap).AddRoad
```

Type-aliasingen gjør at for eksempel `RailLine` blir behandlet som om det sto "Road", mens disse metode-variablene gjør at vi kan kalle på dem med de nye navnene. Selv om aliasing tilsynelatende tilbyr lignende funksjonalitet som endring av navn i GoPT, er det ganske så begrenset, og vi kan for eksempel heller ikke benytte oss av dette for å endre navn på structers felter. Under følger et kode-utdrag for testing av tognettet med Go2, tilsvarende utdraget med GoPT over:

```
var railMap RailMap
railMap.graph = &Graph[Place, TravelTime]{}

aberdeen := TrainStation[Place]{item: Place { name: "Aberdeen",
    latitude: 57.14, longitude: -2.10 } }
...
AddCity(&railMap, &aberdeen)
...
```

```
aberdeenToBrussel := &RailLine[Place, TravelTime]{edge: &Edge[Place,
    TravelTime]{from: &aberdeen, to: &brussel}, meansOfTransport:
    &Train[Place, TravelTime]{"train"}, km: 1220.0}
aberdeenToBrussel.edge.edgeCost = TravelTime{cost:
   aberdeenToBrussel
      .meansOfTransport
         .TimeBetweenTwoPlaces(aberdeenToBrussel)}
...
AddRailLine(&railMap, aberdeenToBrussel, brusselToAberdeen)
...
path, _ := railMap.graph.ShortestPath(aberdeen, elche)
...
fmt.Println("\n", railMap.CalculateShortestRoutes(&aberdeen))
```

Forskjeller mellom GoPT og Go2

Vi skal nå se litt nærmere på forskjellene mellom de to eksemplene vi har vært igjennom.

I situasjoner der vi i GoPT-eksempelet brukte "addto"-setningen kunne vi i Go2-eksempelet få til noe lignende ved å deklarere en ny struct, for så å bruke komposisjon med den opprinnelige structen. Dog var dette kun mulig dersom vi også endret navn på structen eller grensesnittet som skulle utvides. Dersom vi ikke hadde endret navn, ville dette bydd på problemer, siden det å deklarere en ny type i dette tilfellet ville ført til "redeklarasjons"-feil. Med andre ord var vi i praksis avhengig av å endre navn på den nye structen eller grensesnitt for å få til noe lignende som "addto"-setningen.

Når det gjaldt mekanismen for endring av navn i GoPT, kunne vi med Go2 til en viss grad benytte oss av aliasing, selv om dette var veldig begrenset. I praksis kunne vi endre deklarasjons-navn på structer og grensesnitt, samt noen metoder. Ellers var vi nødt til å deklarere nye typer og bruke komposisjon, eventuelt deklarere nye funksjoner og kalle på dem opprinnelige. Det at vi ikke har mulighet til å endre navn på structers felter med Go2 gjør at vi må velge mellom å bruke de opprinnelige felt-navnene, eller deklarere helt nye structer (med nye felter) og dermed ikke gjenbruke den opprinnelige structen. Som nevnt tidligere valgte vi i Go2-eksempelet å la feltene hete det som opprinnelig var definert, slik at vi i alle fall kunne gjenbruke en del av koden. Dog begrenser dette abstraksjonen fra tekniske implementasjonsmessige aspekter til de konkrete bruksområdene. For et geografisk nettverk vil det antageligvis være mer presist for dette konkrete scenarioet å bruke begreper som "travelTime" og "places", slik vi kunne med GoPT, og ikke "edgeCost" og "nodes" som vi måtte bruke i Go2-eksempelet. Riktig navngivning er viktig for forståelsen av programmer [13].

For mekanismene for generiske typeparametere fant vi ut at Go2-varianten krevde ganske signifikant mye mer tekst. På disse små utdragene vi har med i denne teksten er det ikke sikkert at det er like lett å se, men i de fulle eksemplene var det veldig tydelig at vi ofte var nødt til å skrive om igjen den samme koden. For eksempel hver gang vi skulle opprette en ny instans av en struct som inneholdt generiske typeparametere, var vi nødt til å konkretisere alle disse uten mulighet for en form for type-inferens. Et eksempel kan være opprettelsen av en toglinje der vi i én instans av `RailLine` var nødt til å konkretisere seks typeparametere:

```
aberdeenToBrussel := &RailLine[Place, TravelTime]{edge: &Edge[Place,
    TravelTime]{from: &aberdeen, to: &brussel}, meansOfTransport:
    &Train[Place, TravelTime]{"train"}, km: 1220.0}
```

Det samme var vi nødt til å gjøre for hver toglinje, slik at for tolv strekninger (seks frem og tilbake) som vi hadde i eksempelet, var vi nødt til å konkretisere 72 ganger. Til sammenligning konkretiserte vi kun én gang med påkrevde typer, noe som skjedde i instansieringen:

```
ptinst GeoNetworkTemplate with
   MeansOfTransport <= Train,
   GeoLocation <= Place, ...
...
aberdeenToBrussel := &RailLine{from: &aberdeen, to: &brussel,
    meansOfTransport: &Train{"train"}, km: 1220.0}
```

Innen de litt over 300 kodelinjene Go2-eksempelet brukte, måtte vi rundt 100 ganger angi konkretiseringer med klammeparenteser. Inne i disse parentesene var det ofte flere typer, slik at det totale antallet var på rundt 200. Antall konkretiseringer i GoPT-eksempelet var 3, noe som er over 66 ganger så få som i Go2-eksempelet!

Da vi lagde disse eksemplene, gjorde vi dette i små omganger, slik at vi gradvis la til flere generiske typeparametere. Det vi da erfarte var at kun små endringer eller utvidelser av generiske typeparametere krevde ekstremt mange endringer i koden på Go2-eksempelet. Dersom vi for eksempel i `Node` skulle lagt til en ny generisk typeparameter, måtte vi endret alle stedene denne structen ble brukt, og også alle stedene den indirekte ble brukt. Et eksempel på indirekte bruk av `Node` kan vi se i `Edge`, som lagrer objekter av `Node`. Dersom `Node` fikk en ny generisk typeparameter, ville vi nå vært nødt til å ta høyde for dette i `Edge` også, i tillegg til alle stedene der `Edge` enten direkte eller indirekte ble brukt. Dette gjorde det også ganske vanskelig og tungvint, siden en liten endring eller tillegg ville føre til flere hundre feilmeldinger.

Det å introdusere en ny påkrevd type i `Node` ville i praksis ikke innebåret noe mer kodeskriving enn deklarering av selve typen, og eventuelt konkretisering av typen i eksisterende instansieringer. Dette er altså i sterk kontrast til Go2, hvor det ikke bare hadde vært mye ekstra kode-skriving, men at det også hadde vært vanskelig å få riktig. Eksemplene vi har brukt i denne oppgaven er også forholdsvis små, og dette ville nok vært enda tydeligere med mange ulike generiske typer, og i hvert fall dersom det hadde vært avhengigheter mellom dem.

For å forsøke å tallfeste forskjellene som ble nevnt over, altså ved å introdusere en ny generisk typeparameter for `Node`, lagde vi en alternativ versjon av eksemplene der structen `Node` nå har et nytt felt som er typet som den generiske typeparameteren `NewType`. På [32] har vi lastet opp resultatene fra "git diff"[1] mellom de nye eksemplene og de opprinnelige. Dette viser altså alle nødvendige tillegg og endringer som krevdes for å få dette til.

Det var veldig kurant å utvide med den nye typeparameteren i GoPT-eksempelet siden dette kun innebar å lage en ny påkrevd type, legge til det

[1] Kommandoen "git diff" vil i vårt tilfelle sammenligne de to kode-eksemplene, med og uten den nye typeparameteren, og skrive ut alle kodelinjer som er nye eller endret

nye feltet i `Node`, deklarere en struct for konkretisering og spesifisere konkretiseringen i "ptinst"-setningen. Dette innebar 8 nye linjer med kode, og én endring i en eksisterende kodelinje.

Når det gjelder Go2-varianten, så beviste dette på mange måter at det å utvide med en ny generisk type er vanskeligere og mer tidkrevende. Selve deklarasjonene av `Node` og skranken til den nye typeparameteren var naturlig nok omtrent likt som GoPT-varianten. Dog måtte vi endre kodelinjer på alle steder som direkte eller indirekte brukte `Node`, noe som i dette eksempelet innebar omtrent alle struct- og metode-deklarasjoner. Sammenlignet med GoPT-versjonen krevde den nye typeparameteren omtrent like mange *nye* kodelinjer (én ekstra for Go2), men når det gjaldt antall linjer som måtte endres så var dette 70 flere for Go2-varianten. Når vi tar med at det kun var én linje for GoPT, så vil dette si at det faktisk også var 70 *ganger* flere kodelinjer som måtte endres i Go2-varianten! I tillegg skal det nevnes at det ofte var flere endringer innen samme kodelinje i Go2-varianten, og faktisk opptil 4 endringer på samme kodelinje. Det vil si at det totale antallet av endringer som krevdes var ganske mye mer enn 70.

Et viktig poeng som vi oppdaget var at vedlikeholdbarheten og generell lesbarhet av programmet var ganske mye dårligere i Go2-varianten sammenlignet med GoPT-varianten. Siden Go2-varianten krevde så mange konkretiseringer, og dermed ekstra mye tekst, var det også vanskeligere som leser av programmet å faktisk forstå hva koden innebar. I tillegg som vi har vært inne på så førte det til at det å vedlikeholde programmet, samt å utvide det, var mye mer tungvint med bruk av Go2.

6.4 Hvordan passer Pakkemal-mekanismen med Go?

Hovedmålet for denne oppgaven var å undersøke og prøve å finne ut av hvordan Pakkemal-mekanismen passer med Go, og hvilke tilpasninger av mekanismen som vil være nødvendige for å kunne passe et slikt språk. Vi har i denne oppgaven vist hvordan vi har tilpasset Pakkemal-konsepter slik at de passer bedre til språket. Dersom vi ikke hadde gjort dette, ville helt klart mekanismen passet nokså dårlig, siden den opprinnelig er utviklet for programmeringsspråk som hovedsakelig bygger på objektorientert programmering. Likevel vil vi nok si at vi har fått implementert Pakkemal-konseptene uten å endre veldig mye på selve formålene til konseptene, og det de skal tilby.

Syntaks

Når det gjelder syntaksen vi har definert i GoPT, så er denne ganske lik det som opprinnelig var definert for Pakkemal-mekanismen. Hvis vi ser på konsepter som pakkemal-deklarasjoner, instansieringer og "addto"-setningen så er syntaksen som er brukt i stor grad lik. Disse konseptene, og mekanismen generelt, skiller seg ganske mye fra vanlige Go-konsepter, og det vi nå har implementert er nokså vanskelig å få til med bruk av vanlig Go-kode. Derfor valgte vi å definere eksplisitt nye konstruksjoner og konsepter, i stedet for å forsøke å putte de inn i allerede eksisterende konsepter i Go. Dersom vi for eksempel skulle implementert støtte for "addto"-setningen uten å eksplisitt definere nye konstruksjoner,

kunne vi kanskje i stedet videreutviklet komposisjon-mekanismen i Go. Likevel vurderte vi det slik at det var bedre å lage et klart skille mellom hva som var nytt ved introduksjonen av Pakkemal-mekanismen, og hva som allerede var eksisterende i språket.

Ved en reell implementasjon ville vi nok dog undersøkt nærmere om det ville vært fornuftig å definere syntaksen mer lik eksisterende konsepter i Go. Utviklerne av Go ønsker som nevnt i kapittel 3 å ha et så lite og kompakt språk som mulig, blant annet ved å ha få nøkkelord. Kanskje kunne vi derfor forsøkt å flette denne nye syntaksen mer inn med eksisterende konstruksjoner. Om det er mulig å definere en syntaks som samtidig blir fornuftig er vanskelig å si, spesielt siden disse konseptene i utgangspunktet er nokså langt unna konsepter i språket fra før. Det kan også være at den beste løsningen uansett er å ha en syntaks som fører til flere nøkkelord og at språket blir større, men som til syvende og sist fører til en enklere og mer ryddig kode.

Uavhengig av hvordan syntaksen *burde* være fra et "språk-design"-perspektiv, så er det slik at vi kunne definere den syntaksen vi selv ønsket, uten at dette gikk på bekostning av allerede eksisterende konstruksjoner som videre førte til uønskede sideeffekter. Dette er positivt siden det tyder på at det i praksis er få begrensninger for hva vi har mulighet til å definere, og at vi dermed står nokså fritt til å utforme disse nye konstruksjonene slik vi selv ønsker. Antageligvis kommer dette av at språket fra før er nokså lite og konsist, slik at det vil forekomme få konflikter ved introduksjon av nye konsepter.

Semantikk

Når det gjelder tilpasninger av Pakkemal-mekanismen som var nødvendig for å kunne implementere det i Go, vil vi nok kunne si at disse tilpassingene var hyppigere og større for semantiske aspekter enn for syntaktiske. En av grunnene til dette er at språket ikke har klasser og objektorientering slik som i Java, men i stedet structer der det ikke er arv slik vi kjenner fra objektorienterte språk. I tillegg har Go en annen tilnærming til grensesnitt, der kompatibilitet ikke baseres på nominelle bindinger, men i stedet strukturen til typen.

Deklarering av pakkemaler og instansiering av disse er konseptuelt nokså likt i GoPT og i den opprinnelige definisjonen. Innholdet i pakkemaler er i stor grad en samling av deklarasjoner, og instansiering handler hovedsakelig om å ta kopi av innholdet og å "sette" disse inn der instansieringen finner sted. Selv om innholdet i pakkemaler er ulikt fra denne versjonen og den opprinnelige, vil vi si at hovedidéen og formålet i stor grad er den samme. Dette med å kopiere innholdet til pakkemaler når vi instansierer er et viktig punkt for å videre kunne bruke andre Pakkemal-konsepter på pakkemalen, og dette kunne vi nokså enkelt løse i Go ved å dyp-kopiere subtrær av AST-et.

Det å utvide typer med bruk av "addto", vil nok antageligvis være viktigere i språk som Java. I slike språk vil for eksempel alt av innhold til klasser være definert inne i klasse-kroppen, noe som vil være tilsvarende for utvidelse av klasser der alt innhold vil være definert "addto"-kroppen. Go har structer i stedet for klasser, og i deklarasjons-kroppen til slike typer er det i praksis kun felter. Dette gjør at "addto"-setningen ikke er like nødvendig i dette språket, og må kun bli brukt dersom vi ønsker å utvide structer med nye felter eller overskrive metoder og funksjoner. Konstruksjonen brukes også dersom vi vil utvide grensesnitt og påkrevde typer.

Det at Go sjekker grensesnitt på en strukturell måte var med på å gjøre det ganske nyttig å kunne utvide grensesnitt. Siden det i all hovedsak er innholdet og strukturen som avgjør kompatibilitet, blir den resulterende mekanismen veldig fleksibel. Samtidig som dette gjør mekanismen enda mer fleksibel, vil kombinasjonen av strukturell typesjekking og utvidelse eller endring av navn på metoder i grensesnitt, kunne føre til noen ekstra utfordringer. Dersom for eksempel et objekt av en struct er typet som et grensesnitt, og vi så endrer navnene til metode-signaturene i grensesnittet, kan dette føre til at bindinger som opprinnelig var gyldige ikke lenger er det. Siden det ikke vil være noen nominelle bindinger mellom grensesnittene og structene, kan det også være vanskeligere å på forhånd se at disse endringene av navn kan føre til slike feil. Dette kan også føre til at feilmeldingene blir dårligere og vanskeligere å forstå. I implementasjonen av GoPT valgte vi løse dette problemet ved å lage eksplisitte sjekker mot disse scenarioene, slik at feilmeldingene ble tydelige. Selv om kombinasjonen av strukturell typesjekking og endring av grensesnitt introduserer noen nye utfordringer, slik at vi blant annet må definere noen ekstra sjekker, vil det i det store og hele kunne tilby enda mer fleksibilitet, noe som gagner selve Pakkemal-mekanismen slik vi ser det.

Pakkemal-konseptet som tillater å sammeflette structer og grensesnitt passer ganske godt med Go og språkets egne konsepter. Denne sammenflettingen av typer kan minne litt om komposisjon slik vi kjenner det i språket fra før, bare enda litt kraftigere og mer fleksibelt. Med bruk av komposisjon i Go kan vi "samle" flere typer i én, og dermed for eksempel benytte oss av all funksjonalitet fra alle de forskjellige typene. Dog kan vi jo med GoPT i tillegg til å sammenflette typer også bruke andre Pakkemal-konsepter videre, slik at vi har enda større fleksibilitet for hvordan den resulterende typen vil ende opp. En sammenfletting av typer i GoPT fører til at ender vi opp med kun én enkelt type, noe som ikke er tilfellet med bruk av komposisjon i Go der vi egentlig bare får referanser til de opprinnelige typene. Dersom en type som har brukt komposisjon har duplikater, for eksempel flere funksjoner eller attributter med samme navn, vil være nødt til å håndtere de opprinnelige typene, og også hva som er innholdet til typene. Med andre ord er vi nødt til å ta hensyn til de ulike typene også etter vi har "samlet" dem. En slik problemstilling vil vi ikke støte på i GoPT, siden vi ikke tillater duplikater, og vi faktisk bare har én type etter en sammenfletting.

Et konsept som overskriving av metoder ("overriding") er noe som typisk er definert i objektorienterte språk. Go er som nevnt tidligere ikke et rent objektorientert språk, og dette med å kunne overskrive metoder er veldig begrenset. For å få til noe lignende må vi bruke en kombinasjon av grensesnitt og komposisjon. La oss anta at vi har et struct-objekt `i` som er typet som et grensesnitt `I`, og at vi bruker dette objektet til å kalle på en metode `F`.

```
type I interface {
   F() string
}

type A struct {}

func (a A) F() string {
   return "Opprinnelig metode"
}
```

```
func main() {
   var i I = A{}
   fmt.Println(i.F())
}
```

I koden over vil funksjonen `main` skrive ut teksten "Opprinnelig metode". Dersom vi skal få til noe lignende som overskriving av metoden `F` i eksempelet over, kan vi deklarere en ny struct som bruker komposisjon med den opprinnelige, og deretter definere en metode med samme navn som metoden som ble kalt. Hvis vi nå i funksjonen "main" endrer objektet `i` til å være av structen `B`, vil teksten "Ny metode" bli skrevet ut i stedet:

```
type I interface {
   F() string
}

type A struct {}

func (a A) F() string {
   return "Opprinnelig metode"
}

type B struct {
   A
}

func (b B) F() string {
   return "Ny metode"
}

func main() {
   var i I = B{A{}}
   fmt.Println(i.F())
}
```

Overskriving i eksempelet over er ganske annerledes fra måten det fungerer med bruk av Pakkemal-mekanismen, og måten vi kan "bytte" ut en funksjonsimplementasjon med en annen i GoPT er nær sagt umulig med vanlig Go-kode. Vi kommer altså ikke unna det å deklarere en ny struct og endre typen til objektene der disse blir brukt, slik som vist i kode-eksempelet over.

Selv om Go fra før ikke har mekanismer for overskriving slik vi definerte det i GoPT, var det kurant å implementere det i språket.

Det å kunne kalle på den opprinnelige metoden etter en overskriving, altså med bruk av ordet `tsuper`, er noe som antageligvis er passende og nyttig i Go. Selv om konteksten for overskriving av metoder i eksempelet over er nokså ulikt fra GoPT, vil vi også her kunne aksessere den "opprinnelige" metoden. Siden grensesnitt-variabler kun har tilgang på metoder, kunne vi dog ikke aksessert metoden direkte via variabelen `i`, siden dette vil kreve å kalle på metoden via aksessering av `A`. Derfor må vi caste variabelen til `B`:

```
func main() {
   var i I = B{A{}}
   b, _ := i.(B)
   fmt.Println(b.A.F())
}
```

Vi ser i eksempelet over at det altså er mulig å få til noe lignende som det `tsuper` i GoPT gjør. Dog er vi nødt til både å caste og kalle på metoden via structen som vi bruker komposisjon med. Dette er til forskjell fra GoPT der vi kun trenger å spesifisere `tsuper` foran metode-kallet, noe som gjør at bruken av dette blir mer fleksibelt og kan brukes i flere sammenhenger. For å få til noe lignende med bruk av vanlig Go, blir vi altså begrenset som vist over.

Generiske typeparametere

I 6.3 ble det vist to eksempler der vi benyttet oss av GoPT-mekanismen for generiske typeparametere, nemlig påkrevde typer. I eksempelet som omhandlet toglinje-nettverk ble det også vist hvordan dette kunne sett ut ved å bruke Go2, og vi sammenlignet også de to ulike variantene. Fra sammenligningen kunne vi se at påkrevde typer i dette eksempelet antageligvis var mer lesbart, og også enklere å vedlikeholde og videreutvikle.

Dog må det nevnes at påkrevde typer er designet for Pakkemal-mekanismen, slik at kombinasjonen av påkrevde typer og bruk av andre Pakkemal-konsepter i et slikt eksempel i utgangspunktet burde fungere godt sammen. Vi må også nevne at per dags dato er nok Go sitt forslag for generiske typeparametere bedre egnet som en generell mekanisme for generiske typeparametere i språket. Dette kommer av at denne for det første er mer ferdigstilt, og at vi kan konkretisere med alle slags typer. For det andre vil nok Go2-mekanismen være litt smidigere i tilfeller der vi for eksempel kun ønsker en liste som inneholder generiske typeparametere. Slik påkrevde typer er definert nå ville vi måtte deklarert en pakkemal med denne listen, for senere å instansiere den. Dog er det ingenting i veien for å videreutvikle påkrevde typer slik at vi for eksempel ikke er bundet opp til kun bruk i pakkemaler, og at vi kan konkretisere med alle slags typer. Dersom vi hadde sørget for dette, ville det nok ikke vært like klart hvilken mekanisme som er best som en generell mekanisme i språket. Når det kommer til hvilken mekanisme som er best for generiske typeparametere i pakkemaler, og kombinasjon med andre Pakkemal-konsepter, er vår erfaring fra arbeidet med denne masteroppgaven at påkrevde typer er en mye bedre egnet mekanisme for dette enn Go sitt eget forslag.

Vi kan også si at påkrevde typer som et konsept passer ganske godt sammen med Go. For eksempel er måten vi definerer skranker omtrent identisk for påkrevde typer og Go sitt eget forslag. Kroppen til en påkrevd type er i denne versjonen lik som i grensesnitt. Dette kan for eksempel utnyttes i en reell implementasjon, der vi kan gjenbruke typesjekking for grensesnitt når vi skal sjekke om konkrete typer faktisk oppfyller kravene til påkrevde typer. I denne oppgaven har vi også utnyttet dette der vi har gjenbrukt typesjekking-kode for grensesnitt når vi skal typesjekke påkrevde typer.

Kapittel 7

Konklusjon og videre arbeid

I denne masteroppgaven har vi sett på hvordan Pakkemal-mekanismen kan utspille seg i programmeringsspråket Go, og at dette kan bidra til gjenbruk av deklarasjoner og strukturer på en fleksibel måte. Til tross for at denne mekanismen i utgangspunktet er designet for objektorienterte språk, har vi vist at konseptene som er definert er såpass fleksible at de også vil kunne gi en nytteverdi i andre typer språk. Vi skal i dette kapittelet forsøke å konkludere prosjektet og fremlegge forslag til videre arbeid på dette feltet.

7.1 Konklusjon

Vi har i denne masteroppgaven videreutviklet en preprosessor der vi har implementert støtte for Pakkemal-mekanismen. Selve implementasjonen har vært en stor del av oppgaven, noe som har innebåret å endre eller legge til rundt 5000 kodelinjer, og koden kan lastes ned fra [33]. Resultatet av dette, implementasjon av språket GoPT, støtter alle konseptene slik de ble vist i kapittel 4. Dette inkluderer blant annet pakkemal-deklarasjoner, instansieringer av disse, utvidelse og sammenfletting av typer, overskriving av funksjoner, samt en mekanisme for generiske typeparametere.

Mekanismen vil som første steg i preprosesseringen gjøre en fullstending syntaktisk analyse, selv om denne analysen kan inneholde feil og mangler, der den skriver ut passende feilmeldinger dersom et innsendt program er syntaktisk ugyldig. Videre vil den gjøre en semantisk sjekk av pakkemal-deklarasjoner, slik at vi kan være sikre på at disse isolert sett er semantisk gyldige. Etter dette gjør vi de nødvendige modifikasjonene på AST-et for å støtte Pakkemal-konseptene, og vi vil deretter igjen gjøre en ny semantisk analyse som sørger for at det resulterende programmet ikke inneholder typefeil. Som siste steg bruker preprosessoren det nye AST-et til å skrive programmet til en ny fil med offisiell, gjeldende Go-syntaks. Den resulterende filen vil så kunne kompileres av en vanlig Go-kompilator.

Denne utvidelsen av Go gjør at vi har en mekanisme som tillater å gjenbruke samlinger av deklarasjoner, og samtidig kan tilpasse disse videre for spesifikke formål. Dette kan føre til at en utvikler heller velger å gjenbruke kode i stedet for å utvikle strukturer og konstruksjoner fra bunnen av. En av styrkene til mekanismen er at vi vil kunne slippe å repetere den samme koden om igjen.

Dette gjelder spesielt i scenarioer der vi bruker påkrevde typer i pakkemaler. I 6.3 ble det for eksempel vist at de samme konkretiseringene måtte gjøres 66 ganger oftere med bruk av vanlig Go (inkludert ny syntaks for generiske typeparametere) enn det som var tilfellet med bruk av GoPT.

Pakkemal-mekanismen i Go kan også være nyttig for å abstrahere bort detaljer knyttet til implementasjon. Konsepter som å kunne endre på navn, utvide typer og konkretisere påkrevde typer er alle redskaper som er del av en abstraksjonsprosess. Dette gjør at en utvikler kan flytte fokus bort fra underliggende strukturer og detaljer, og generelt hvordan større konstruksjoner er bygget opp, til å heller fokusere på hva disse konstruksjonene tilbyr av funksjonalitet. Dersom vi har tilpasset disse konstruksjonene og konseptene med for eksempel endring av navn, vil forhåpentligvis også lesbarheten være bedre, siden det på denne måten blir tydeligere hva programmet faktisk gjør og fører til.

I innledningen listet vi opp fire konkrete mål for hva vi ønsket å oppnå med dette prosjektet. Det første punktet gikk på å undersøke hvordan Pakkemal-mekanismen fungerer i et språk som Go, og hvilke tilpasninger som vil være fornuftig å gjøre. Dette målet oppnådde vi i stor grad ved å definere språket GoPT, som er en kombinasjon av programmeringsspråket Go og konsepter fra Pakkemal-mekanismen.

Videre ønsket vi å utforske hvordan Go sitt typesystem ville påvirke en introduksjon og implementering av Pakkemal-mekanismen i Go. Som vi har vært inne på har dette bydd på noen ekstra utfordringer, men i det store og hele gitt en enda større fleksibilitet, der for eksempel bruk av Pakkemal-konsepter på grensesnitt har virket fornuftig og nyttig. Antageligvis vil strukturell typesjekking av grensesnitt generelt sett føre til at aspekter som tilpassing og utvidelse av grensesnitt i det hele tatt blir mer aktuelt for en utvikler å benytte seg av sammenlignet med Pakkemal-mekanismen i for eksempel Java, siden vi ikke vil måtte ta høyde for nominelle avhengigheter.

Det tredje punktet, som gikk på å forsøke å implementere en mekanisme for gjenbruk av kode som tilbyr tilsvarende funksjonalitet som Pakkemal-mekanismen, ble oppnådd ved å implementere GoPT ved å videreutvikle en preprosessor.

Det siste punktet gikk på det å få en forståelse for om Pakkemal-mekanismen kan gi en nytteverdi for et språk som ikke hovedsakelig bygger på objektorientert programmering, slik som Go. Vi mener dette målet er oppnådd, siden introduksjonen av Pakkemal-mekanismen i Go har gitt språket en ny måte for å fleksibelt kunne gjenbruke deklarasjoner og konstruksjoner, som både kan bidra til økt gjenbruk av kode og også til en større grad abstrahere bort detaljer knyttet til implementasjon. Sammenligningene med GoPT- og Go2-kode i 6.3 viste at aspekter som for eksempel endring av navn på "dype" datastrukturer ikke var mulig med bruk av Go2 uten å deklarere nye typer. Dersom vi med bruk av Go2-kode ville få til noe lignende som "addto"-setningen, ville vi også vært nødt til å deklarere nye typer, og det ville i tillegg fordret at vi måtte gitt nytt navn til den nye typen. Sammenligningene viste at vi til en større grad kunne gjenbruke eksisterende deklarasjoner og konstruksjoner med GoPT enn det som var mulig med Go2. En annen nytteverdi Pakkemal-mekanismen, og særlig mekanismen for generiske typeparametere, kunne gi, var å til enda større grad abstrahere bort detaljer. Ved å for eksempel kunne endre navn på deklarasjoner ble koden mer presis for de konkrete scenarioene, siden navnene dermed relaterte til de konkrete bruksområdene. Det at vi slapp mange gjentakelser av

generiske skranker bidro også til å fjerne "forstyrrende" elementer, og sørget for mer lesbar, og ikke minst mer vedlikeholdbar, kode.

Oppsummert vil vi si at vi med denne oppgaven i stor grad har oppnådd målene vi satte oss i innledningen.

7.2 Videre arbeid

I denne versjonen av GoPT har vi ikke hatt tid til en grundig test-prosess, slik at det er sannsynlig at programmet inneholder enkelte feil. Likevel har vi laget noen større eksempler uten problemer, så dette vil nok hovedsakelig være små problemer som er nokså enkle å rette opp i. I tillegg kunne nok koden hatt godt av en refaktorering, der vi også kunne forsøkt å gjøre den enda mer effektiv. Det ville også vært fordelaktig å benytte Go sitt eget rammeverk for regresjonstester, og definert spesifikke tester for de implementerte Pakkemal-konseptene. På denne måten ville det vært enklere å forsikre seg om at nye endringer og utvidelser i koden ikke ville føre til utilsiktede sideeffekter på allerede implementerte konsepter. De nevnte aspektene over kunne inngått i en prosess der vi ville forsøkt å utvikle en mer robust og produksjonsklar implementasjon.

Under vil vi kort gå inn på noen konkrete forslag for videre utvikling av GoPT:

- **Sammenfletting av typer fra samme pakkemal**

  Implementasjonen av GoPT tar ikke høyde for å sammenflette typer fra samme pakkemal. Det er sannsynlig at dette er noe som vil kunne være nyttig med denne mekanismen. Dersom vi forsøker dette slik mekanismen er nå, vil det resulterende programmet ende opp med duplikater. Antageligvis er dette noe som er relativt enkelt å implementere støtte for.

- **Bruk av Pakkemal-konsepter på andre typer deklarasjoner**

  Det å kunne bruke Pakkemal-konsepter på for eksempel variabel- og konstant-deklarasjoner av basistyper og sammensatte typer er noe som antageligvis ville vært nyttig. Vi har allerede sett i 6.3.2 at bruk av konstant-deklarasjon er nyttig. Mekanismen støtter altså deklarasjoner av dette, og de kan også brukes på vanlig måte. Dog vil vi for eksempel ikke kunne endre navn på disse deklarasjonene, og det er også sannsynlig at det ville være nyttig å kunne bruke andre Pakkemal-konsepter på disse typene. Det ville nok også være fornuftig å utforske andre konsepter som er implementert i Go, og undersøke om bruk av Pakkemal-konsepter på disse kunne gitt en nytteverdi. Et eksempel på dette, som vi har nevnt tidligere, kan være Go sin støtte for eksplisitte datastrukturer for JSON.

- **Større fleksibilitet for påkrevde typer**

  Påkrevde typer er i denne versjonen begrenset kun til å kunne konkretiseres med structer. Likevel vil det være scenarioer der det også ville vært nyttig å kunne konkretisere med for eksempel grensesnitt for å tilby enda større fleksibilitet med tanke på hvilke typer som blir brukt etter instansiering og konkretisering. I tillegg burde det undersøkes videre om påkrevde typer også burde kunne konkretiseres med basis-typer, slik som mange andre mekanismer for generiske typeparametere tilbyr. Kanskje

kunne dette ført til at påkrevde typer også kunne fungert som en generell mekanisme for generiske typeparametere.

- **Større eksempler**

  Eksemplene vi har brukt i denne oppgaven er nokså små der de største strekker seg opp mot 400 kodelinjer. Det hadde derfor vært interessant å implementere for eksempel noen større åpne kildekode-bibliotek med bruk av GoPT, og sett hvordan språket ville håndtert dette. I tillegg kunne det vært nyttig å utforske om Go sitt eget standard-bibliotek kunne blitt forenklet dersom vi hadde antatt at vi hadde Pakkemal-mekanismen fra start av.

- **Teste mekanismen på forsøkspersoner**

  Vi har til nå ikke testet hvordan andre personer oppfatter mekanismen. Det ville vært interessant å for eksempel teste dette opp mot studenter og Go-utviklere. For studenter som ikke har erfaring med Go fra tidligere ville det først og fremst vært nyttig å finne ut om GoPT-kode ble oppfattet som mer lesbart enn tilsvarende kode med bruk av vanlig Go. For Go-utviklere ville det vært spesielt interessant å finne ut om mekanismen ville ført til mer gjenbruk av kode der de ellers hadde definert nye deklarasjoner og konstruksjoner.

Bibliografi

[1] Amarjeet og Jitender Kumar Chhabra. "Improving package structure of object-oriented software using multi-objective optimization and weighted class connections". *Journal of King Saud University - Computer and Information Sciences* 29.3 (2017), s. 350. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2015.09.004.

[2] Eyvind W. Axelsen og Stein Krogdahl. "Groovy Package Templates: Supporting Reuse and Runtime Adaption of Class Hierarchies". *SIGPLAN Not.* 44.12 (okt. 2009). ISSN: 0362-1340. DOI: 10.1145/1837513.1640139.

[3] Eyvind Axelsen og Stein Krogdahl. "Adaptable generic programming with required type specifications and package templates". *AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development* (mar. 2012). DOI: 10.1145/2162049.2162060.

[4] Eyvind Axelsen og Stein Krogdahl. "Package Templates: A Definition by Semantics-Preserving Source-to-Source Transformations to Efficient Java Code". *ACM SIGPLAN Notices* 48 (sep. 2012). DOI: 10.1145/2371401.2371409.

[5] Eyvind Axelsen mfl. "Challenges in the Design of the Package Template Mechanism" (jan. 2012), s. 9. DOI: 10.1007/978-3-642-35551-6_7.

[6] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, 2002, s. 15. ISBN: 0321146530.

[7] Gilad Bracha mfl. "Making the Future Safe for the Past: Adding Genericity to the Java Programming Language". *SIGPLAN Not.* 33.10 (okt. 1998), s. 183–200. ISSN: 0362-1340. DOI: 10.1145/286942.286957.

[8] Russ Cox. *Eleven Years of Go*. 10. nov. 2020. URL: https://blog.golang.org/11years (sjekket 20.04.2021).

[9] J. T. Davie og R. Morrison. *Recursive Descent Compiling*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 1982, s. 15–17. ISBN: 0470273615.

[10] Alan A.A. Donovan og Brian W. Kernighan. *The Go Programming Language*. 1. utg. Addison-Wesley Professional, 2015, s. 41–44, 283–300. ISBN: 0134190440.

[11] Andrew Gerrand. *Go maps in action*. 6. feb. 2013. URL: https://blog.golang.org/maps (sjekket 06.03.2020).

[12] James W. Hooper og R.O. Chester. *Software Reuse: Guidelines and Methods*. Springer Science & Business Media, 1991, s. 1. ISBN: 9780306439186.

[13] Einar Høst. "Meaningful Method Names". Ph.d.-avh. Universitetet i Oslo, 2011, s. 13–15. URL: https://www.duo.uio.no/handle/10852/8842.

[14] Eirik Isene. "PT# - Package Templates in C#". Masteroppg. Universitetet i Oslo, 2018. URL: https://www.duo.uio.no/bitstream/handle/10852/62821/ThesisEirikIsene.pdf?sequence=1&isAllowed=y.

[15] *Java Language Keywords*. (u.å.) URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html (sjekket 12.03.2021).

[16] Stein Krogdahl, Birger Møller-Pedersen og Fredrik Sørensen. "Exploring the Use of Package Templates for Flexible Re-use of Collections of Related Classes." *Journal of Object Technology* 8 (nov. 2009), s. 59–85. DOI: 10.5381/jot.2009.8.7.a1.

[17] Paul Leach, Michael Mealling og Rich Salz. *A universally unique identifier (uuid) urn namespace*. RFC 4122. Jul. 2005, s. 2–5. URL: https://www.hjp.at/doc/rfc/rfc4122.html.

[18] John Levine, Tony Mason og Doug Brown. *Lex & Yacc*. 2. utg. O'Reilly, 1992, s. 1–22. ISBN: 9781565920002.

[19] Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. 2012. URL: https://talks.golang.org/2012/splash.article (sjekket 04.03.2021).

[20] Rob Pike. *Go's Declaration Syntax*. 7. jul. 2010. URL: https://blog.golang.org/gos-declaration-syntax (sjekket 05.02.2021).

[21] *pt-compiler*. 2013. URL: https://github.com/uio-jpt/pt-compiler (sjekket 13.05.2021).

[22] Robert W. Sebesta. *Concepts of Programming Languages*. 10. utg. Pearson, 2012, s. 210–214, 302–304. ISBN: 0273769103.

[23] Martin Steffen. "Course Script - INF5110: Compiler construction" (2020), s. 6–7, 27, 81. URL: https://www.uio.no/studier/emner/matnat/ifi/INF5110/v20/slides/slides-script.pdf (sjekket 03.04.2021).

[24] Håkon Stordahl. "BooPT: Implementasjon av Package Templates for Boo". Masteroppg. Universitetet i Oslo, 2011. URL: https://www.duo.uio.no/bitstream/handle/10852/9025/stordahl-master.pdf?sequence=2&isAllowed=y.

[25] Ian Lance Taylor. *A Proposal for Adding Generics to Go*. 12. jan. 2021. URL: https://blog.golang.org/generics-proposal (sjekket 13.03.2021).

[26] Ian Lance Taylor. *Type Parameters Proposal*. 19. mar. 2021. URL: https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md (sjekket 13.05.2021).

[27] *The Go Programming Language*. URL: https://github.com/golang/go (sjekket 13.05.2021).

[28] *The Go Programming Language - go2go branch*. URL: https://github.com/golang/go/tree/dev.go2go (sjekket 13.05.2021).

[29] *The State of the Octoverse*. 2019. URL: https://octoverse.github.com/2019 (sjekket 17.03.2021).

[30] *Why did you create a new language?* (u.å.) URL: https://golang.org/doc/faq#creating_a_new_language (sjekket 05.03.2021).

[31] *Why doesn't Go have implements declarations?* (u.å.) URL: https://golang.org/doc/faq#implements_interface (sjekket 08.03.2021).

[32] Kristian Aadalen. *Example programs for GoPT*. 2021. URL: https://bitbucket.org/kaada/gopt-exampleprograms/src/master.

[33] Kristian Aadalen. *Source code of GoPT*. 2021. URL: https://bitbucket.org/kaada/gopt/src/gopt_prototype_final.