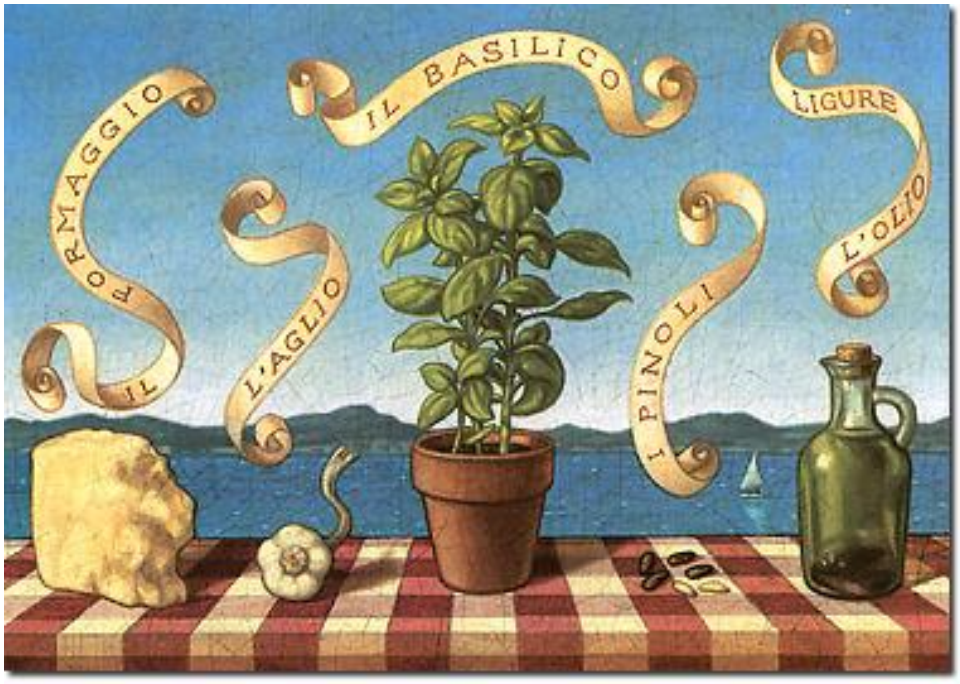
Deklarativ specialisering af objektorienterede programmer



Pesto – et deklarativt sprog til partiel evaluering

Helle Markmann
Maj 2003
Datalogisk Institut
Aarhus Universitet

Tak til Mikkel Ricky for den fantastiske snippet-pakke, direkte Latex-hotline og kommentarer om stort og småt. Og et stort tak til Frokostklubben som har været dagens faste holdepunkt under en ensom proces. Og selvfølgelig tak til min familie, min kæreste Klaus og søn Ask, og ikke mindst til min mor for kommaerne.

Abstract

Partial evaluation is an automatic program transformation that adapts a program to the execution context in which it is used, so that it runs faster or occupies less space. Partial evaluation relies on information from the user, a so-called specialization context, but this context is complicated to write and understand. In order to help the ordinary programmer in writing a specialization context a new language, Pesto, is introduced.

Pesto is a declaration language for specialization of object-oriented programs. It lets the user declare specialization invariants in an intuitive way. The language Pesto is based on *specialization classes* and on the object-oriented paradigm which provides intuitive and simple structuring of the specialization context. Pesto translates specialization classes into a specialization context for a program specializer for Java. Furthermore, the language creates a *specialization module* containing the specialized code generated by the specializer and a *guard* which during runtime chooses which method to use—the specialized or the generic one—depending on the program context.

The complete encapsulation of the declarative language, the specializer and the guard is an elegant solution for an otherwise complicated task.

Indhold

1 Indledning

Efterhånden ses computere overalt. Ikke bare på ethvert skrivebord i ethvert hjem,[1] men også i vores kalendere, vores telefoner og tilmed i vores køkkenredskaber. Computerne bliver mindre og mindre og skal kunne mere og mere. Alle disse små computere skal programmeres, og det skal helst være så let som muligt at gøre uden at man er ekspert i bytekode – og man skal helst gøre det med generelle komponenter og designmønstre. Men objektorienterede programmer og designmønstre giver overhead både i forhold til plads og tid, og kræver derfor mere hukommelse og processorkraft hvilket der ikke altid er så meget af i små computere. Derfor er der brug for en metode til at reducere programmerne og få dem til at køre hurtigere.

En måde at reducere overheadet på er at målrette koden til formålet ved delvist at evaluere den generelle kode inden programmerne køres så koden passer specielt til den enkelte enhed – som om vi havde skrevet koden i et sprog i et lavere niveau eller uden brug af generelle mønstre – koden bliver *specialiseret* til formålet. Denne metode kaldes programspecialisering eller *partiel evaluering*.

Partiel evaluering er et kendt forskningsområde som specielt har været brugt til specialisering af funktionelle sprog. Behovet for specialisering af objektorienterede programmer er opstået i takt med behovet for mindre, generelle komponenter til hardware. Men partiel evaluering har ikke haft en særlig stor udbredelse uden for forskerkredse på trods af at ideen ligger lige for. Problemet er at partiel evaluering er svært at bruge – der er mange ting der skal specificeres. Det rejser behovet for et ekstra lag – en mulighed for let at kunne specificere hvad det er for nogle punkter i programmet der skal specialiseres. Netop dette lag er målet for dette speciale: *At definere og implementere et deklarativt sprog til partiel evaluering af objektorienterede programmer.*

Ud over det interessante i at kunne specialisere programmer så de kører hurtigere og bruger mindre plads, rammer dette speciale et andet af mine interesseområde, brugervenlighed; at gøre et område der tidligere har været utilgængeligt for andre end forskere, tilgængeligt for ikke-eksperter. For at kunne benytte programspecialisering

[1] Bill Gates vision tidligt i Microsofts historie (jeg er ikke bekendt med årstallet).

til at skrive programmer er det naturligvis nødvendigt at kunne programmere, men det er ikke nødvendigt for programmøren at sætte sig ind i andet end nogle grundliggende egenskaber for partiel evaluering. Derudover skal han selvfølgelig lære at bruge det deklarative sprog jeg foreslår, men en af målsætningerne for sproget er netop at det skal være let og intuitivt. Så brugervenligheden har været et vigtigt emne under udviklingen af projektet.

Ideen til at skrive et deklarativt sprog til specialisering er ikke min egen. Eugen-Nicolae Volanschi færdiggjorde i 1998 sin ph.d.-afhandling som resulterede i et sådant sprog kaldet *Specialization Classes* [Volanschi98b]. Mit projekt bygger på Volanchis oprindelige ide, men i en udbygget version. Siden 1998 er der desuden fremkommet nye resultater inden for partiel evaluering af objektorienterede sprog og dermed nye behov til sproget. Jeg har under et to-semesters ophold i Frankrig været så heldig at arbejde i gruppen Compose hvor Volanschi skrev sin afhandling, og det var under dette ophold at jeg blev bekendt med og ikke mindst interesseret i problemstilling.

1.1 Ordvalg

Dette speciale er skrevet på dansk hvilket på dette tidspunkt nok ikke kommer som en overraskelse for læseren. Det er et bevidst og velovervejet valg jeg har taget, og grundene er blandt andet at det har været en interessant proces for mig at arbejde med projektet, og derfor ønsker jeg også at dokumentationen af projektet, det vil sige nærværende speciale, er velformuleret, og det klarer jeg bedst på dansk.

Et klassisk problem der opstår når man vælger at skrive en fagtekst på dansk frem for engelsk, er at mange af de begreber der knytter sig til faget, og nok specielt til datalogi, er engelske. Jeg har bevidst forsøgt at bruge så få engelske begreber som muligt da jeg mener det nedsætter læsbarheden, specielt hvis der er et tilsvarende godt udtryk på dansk hvilket jeg ofte synes der er. Nogle gange er jeg kommet til kort og har enten måtte indse at det engelske begreb er bedre eller konsulteret Dansk Sprognævn enten per telefon til deres spørgelinje eller på deres udmærkede it-ordliste [Sprognævnet03].

Problemet med engelsk i en dansk tekst er blandt andet at det er svært at gradbøje, for eksempel havde jeg brug for det der på engelsk hedder *cross cutting* – hvis jeg bruge begrebet *cross cut* ville skulle bøje ordet til noget i retningen af cross cuttende… I stedet har jeg gjort mig umage med at finde et bedre ord. De steder i specialet hvor jeg har valgt at bruge et mindre kendt dansk begreb, har jeg i parentes skrevet det engelske ord/udtryk.

Udover valget af sprog har jeg foretaget nogle få bevidste valg med hensyn til opsætningen af specialet. Blandt andet har jeg valgt at alt hvad der henvises til uden for

brødteksten er figurer, uanset om indholdet er en tabel. Dette, og andre, valg er foretaget efter anbefaling af TechKnowledge Corp [tec88] som har lavet en grundig brugerundersøgelse af hvad der er mest læservenligt.

Dette speciale er kommateret efter de nye kommaregler som anbefalet af Dansk Sprognævn.

1.2 Læsevejledning

Jeg gør i høj grad brug af henvisninger til udsnit af kode og for at gøre kodeudsnittene så lette at overskue som muligt har jeg valgt at *forskønne* dem i stedet for blot at vise dem i et skriftsnit med fast bogstavbredde som man ofte ser. I kodestumperne vil forskellige kategorier af koden blive skrevet med forskellige skriftsnit. For eksempel skrives nøgleord med **fed** og variabelnavne med *kursiv*, og klasse- og metodenavne types med Helvetica. Når jeg i brødteksten henviser til dele af koden, bruger jeg de samme skriftsnit så en variabel altid er skrevet med kursiv osv.

Når jeg har brug for at illustrere noget med et lille stykke kode gøres det i brødteksten med et *kodestykke* der har et nummer så jeg senere kan referere til kodestykket.

1.3 Resten af specialet

Før jeg går i gang med at beskrive mit bidrag til udbredelsen af partiel evaluering, vil jeg i kapitel 2 gøre rede for nogle af de områder det er nødvendigt at have kendskab til for at få så meget ud af læsningen af dette speciale som muligt – blandt andet vil jeg give en mere grundig beskrivelse af partiel evaluering end den jeg har brugt som motivation i dette kapitel. I kapitel 3 vil jeg give en generel beskrivelse af det sprog jeg har designet og udviklet for at give læseren en god forståelse for sproget inden jeg går i detaljer. Kapitel 4 indeholder en mere detaljeret beskrivelse af sproget, og jeg vil redegøre for og diskutere nogle af de overvejelser jeg har haft under udviklingen af sproget. Derefter vil jeg i kapitel 5 gå endnu dybere ind i systemet og kigge på hvad der ligger under sproget, og hvordan det virker i sammenhæng med partiel evaluering. I kapitel 6 vil jeg diskutere den forskning der foregår i projekter med samme problemområde som dette speciale, og jeg vil beskrive nogle af de perspektiver der er i mit arbejde, det vil sige nogle af de arbejdsområder jeg vil tage fat på hvis jeg skal fortsætte med projektet.

2 Baggrund

Inden jeg går i gang med det egentlige emne for dette speciale, nemlig det deklarative sprog jeg har udviklet, vil jeg bruge dette kapitel på at beskrive de teknologier der har betydning for specialet. Jeg vil begynde i kapitel 2.1 med at gøre rede for *partiel evaluering* som er den teknik der ligger til grund for mit speciale. Partiel evaluering har traditionelt været brugt til funktionelle sprog, og derfor beskriver jeg i kapitel 2.2 partiel evaluering for objektorienterede sprog. Kapitel 2.3 giver et indblik i JSpec som er en partiel evaluator til Java. For at give en god forståelse for hvad det er for en type sprog jeg har udviklet, vil jeg i kapitel 2.4 gøre rede for hvad deklarative og domænespecifikke sprog er. Kapitel 2.5 beskriver en anden programmeringsform som jeg også vil benytte senere i specialet, nemlig aspektorienteret programmering.

2.1 Partiel evaluering

De fleste moderne oversættere genererer optimeret kode. En af de mange optimeringer der benyttes er at udskifte en variabel med dens værdi for at undgå at bruge plads og tid på opslag i tabeller under udførelsen af programmet. Men ved almindelig optimering kan oversætteren kun optimere for de værdier den rent faktisk kender.

Programspecialisering er en anden form for optimering som kan gøre brug af data der ikke nødvendigvis står i et program, men som brugeren af programmet kender. Programspecialisering *specialiserer* programmet med hensyn til en konkret *programomgivelse*. At specialisere et program vil sige at optimere det til en helt specielt brug, for eksempel for nogle bestemte værdier. Disse værdier kan gives direkte fra brugeren eller fra invarianter i programkoden. Programspecialisering kan benytte mere aggressive strategier end optimering ved at bruge denne ekstra information [Schultz02]. En *Partiel evaluator* er et program der automatisk udfører programspecialisering, også kaldet partiel evaluering [Jones93].

Der findes flere former for partiel evaluering, men i dette speciale vil jeg hovedsagelig beskrive *off-line* partiel evaluering [Jones93]. *On-line* partiel evaluering vil blive berørt i kapitel 6.1.

Et simpelt og ofte brugt eksempel på partiel evaluering er givet ved hjælp af eksponentialfunktionen, x^y hvor en basis, x, multipliceres med sig selv y gange. I stedet for x og y bruger jeg nogle mere sigende navne når jeg implementerer eksponentialfunktionen. Basis x kalder jeg *base*, og eksponenten y kalder jeg *exp*. En typisk måde at implementere eksponentialfunktionen på er ved hjælp af en **for**-løkke:

```
result := 1;
for (i := 0; i < exp; i := i+1)                    (1)
   result := result*base;
```

Hvis denne funktion ofte bruges hvor eksponenten *exp* er 3, er følgende kode meget mere effektiv (og ydermere pladsbesparende):

```
result = base*base*base;                           (2)
```

Det sidste kodeeksempel er en *specialisering* af det første med hensyn til programkonteksten $exp = 3$.

Formålet med programspecialisering er at programmet efter specialiseringen kører hurtigere eller optager mindre hukommelse. Forskellen på programspecialisering og oversætteroptimering er at når et program specialiseres, optimeres det ud fra *konkrete* programparametre, som ikke er kendte i selve programmet. Det vil sige at der tilføjes ekstra information som kan bruges under optimeringen (som for eksempel at $exp = 3$). Ved almindelig optimering under oversættelsen af et program, som den der sker i for eksempel C-oversætteren, optimeres programmet *generelt* på oversættelsestidpunktet ved at benytte en eller flere analyser og optimeringsmetoder (for eksempel ved eliminering af død kode, det vil sige kode der aldrig udføres). Programspecialisering og oversætteroptimering er ortogonale, det vil sige at et program kan optimeres efter at det er specialiseret, og specialisering medfører ofte at en bedre optimering kan foretages af oversætteren da programstrukturen ændres, og den ændrede kode kan udløse nye optimeringer der ikke blev udløst i den generiske kode [Schultz02].

Statiske og dynamiske variabler

Princippet ved partiel evaluering er at specialisere programmet med hensyn til *kendte* parametre som $exp = 3$ i ovenstående eksempel. Programmets data opdeles i *statisk* og *dynamisk* data. Data siges at være statisk når det er kendt på specialiseringstidspunktet ($exp = 3$) og dynamisk når det ikke er kendt (*base*). Jo mere data der er kendt, des mere kan programmet specialiseres. Generelt kan det udtrykkes på følgende måde:

> Hvis et program P tager to argumenter, s og d, hvor s er statisk og d er dynamisk, kan en *specialisator*[1] Spec specialisere P med hensyn til s og få det resulterende

[1] En specialisator er et program der automatisk udfører specialisering (*specializer*).

program $P_s(d)$.

Spec tager som input et program og de statiske informationer:

$$\begin{aligned} &\mathsf{Spec}(P, s) = P_s(d) \\ &P_s(d) = P(s, d) \end{aligned} \tag{3}$$

Per definition bevares semantikken af P efter specialiseringen, og P_s giver med d som input det samme resultat som P med d og s som input [Consel93]. Afhængig af strukturen af P, vil P_s køre hurtigere end P da de beregninger der kun afhænger af s, vil blive optimeret væk under oversættelsen af P_s.

Hvordan virker partiel evaluering

De mest brugte transformationer ved specialisering er *constant folding*, *control flow simplification* og nogle typer *strength reduction* [Jones93].

Constant folding Udtryk hvis operatorer vides at være konstante evalueres under oversættelsen.

Control flow simplification Forenkling af kontrolflowet i programmet, blandt andet ved udfoldning af løkker som i det lille eksempel tidligere, og kode der aldrig udføres fjernes.

Strength reduction Når det er muligt og rentablet, udskiftes dyre operationer som multiplikation og division med billigere operationer som subtraktion og addition.

Det er ikke altid en fordel at benytte ovenstående transformationsteknikker ukritisk til at specialisere da det kan medføre dårlig specialisering. Man taler om to former for uheldig specialisering: „overspecialisering“ og „underspecialisering“. Et eksempel på overspecialisering er at et maksimalt løkkeindeks er kendt (statisk), men meget stort, og kroppen af løkken er dynamisk, og dermed vil udfoldelsen af løkken give en uheldig eksplosion i kodemængden. „Underspecialisering“ vil sige at programmet ikke bliver specialiseret nok i forhold til hvad der er muligt, for eksempel hvis strukturen i programmet er uhensigtsmæssig for specialiseringen – det kan være at de statiske dele af programmet er afhængige af de dynamiske [Consel93], for eksempel vil der være forskel på specialiseringen af følgende to udtryk: $(s_1 + s_2) + d$ og $s_1 + (s_2 + d)$ hvor s_1 og s_2 er statiske og d dynamisk. Det første udtryk vil specialisere bedre end det andet da udtrykket i parentesen i det første vil kunne reduceres, mens det i det andet vil være dynamisk hvormed hele udtrykket vil være dynamisk, og koden vil være underspecialiseret.

For at undgå over- og underspecialisering er det nødvendigt at programmøren har kendskab til specialisering, eller blot til nogle få grundliggende egenskaber som de ovenstående når han benytter programspecialisering.

Ofte er det i praksis ikke særlig smart at specialisere et helt program da det ikke er alle programdele der egner sig til at blive specialiseret, jævnfør ovenstående om under- og overspecialisering. Som oftes udvælges et programudsnit (*program slice*) som egner sig til at blive specialiseret. Desuden giver det mere præcis specialisering at udvælge et programudsnit da den programkontekst der angiver hvilke værdier der skal specialiseres for, kan være mere præcis. At specialisere en del af programmet er desuden at foretrække i forhold til den tid det tager at specialisere.

Bindingstidsanalyse

Før et program specialiseres, analyseres det, og programvariablerne deles op i to grupper, de statiske og de dynamiske, og det afgøres hermed hvilken *bindingstid* variablerne har, det vil sige hvornår værdien af variablen kan beregnes. Denne analyse kaldes *bindingstidsanalyse* (*binding time analysis, BTA*) og foregår ved eventuelt flere gennemløb af programmet. De specifikke værdier er ikke interessante under bindingstidsanalysen – det er kun relevant hvorvidt variablerne er statiske eller dynamiske. Den information der opsamles under analysen, overføres til specialiseringsfasen hvor de faktiske værdier benyttes til at generere det specialiserede program. Den specifikke bindingstidsanalyse kan på den måde bruges til forskellige eksakte værdier hvilket er en fordel da analysen typisk er det mest tidskrævende ved programspecialisering. Selve specialiseringen hvor de nu kendte værdier sættes ind i analysen, er knap så tidskrævende, og det kan derfor betale sig at bruge den samme analyse til forskellige værdier.

For kodestykke 1 vil det gælde at variablerne *base* og *result* er dynamiske og *exp* statisk. Analysen vil give information om at løkken kan udfoldes og reduceres, men først i specialiseringsfasen udnyttes at *exp* har værdien 3, og dermed fås den endelige kode som ses i kodestykke 2. Derved er det muligt at bruge analysen til specialiseringer med andre værdier af *exp*.

2.2 Objektorienteret partiel evaluering

Objektorienteret programmering giver programmøren et værktøj til at modellere et „problem fra den virkelige verden“ på en måde så koden afspejler den faktiske model. For at få fuldt udbytte af objektorientering ved især komplekse modeller konstrueres modellen ofte finkornet, det vil sige med mange mindre objekter. Disse objekter kommunikerer med hinanden ved hjælp af virtuelle kald. Det er svært for en oversætter at forudsige hvor et virtuelt kald ender, og derfor kan koden ikke optimeres særligt godt

– koden er ikke sammenhængende. Jo mere sammenhængende kode, jo bedre kan den optimeres af en oversætter.

Nogle af omkostningerne ved objektorienterede programmer kan fjernes ved at bruge partiel evaluering.

Objektorienteret programmering opfordrer i høj grad til genbrug af kode i form af designmønstre [Gamma95] og objektkomponenter. Designmønstre giver en generel objektorienteret skabelon til at implementere et problem. For at løsningen netop skal være så generel som mulig, er programdelene delt ud i uafhængige objekter hvilket fører til mindre sammenhængende og dermed mindre effektiv kode som ikke optimerer særlig godt ved almindelig optimering. Som en opfølgning på designmønstre og det overhead der introduceres ved at bruge dem, findes der tilsvarende mønstre som giver en generel beskrivelse af hvordan man specialiserer nogle af de kendte designmønstre med partiel evaluering. Jeg vil beskrive *Specialization Patterns* [Schultz00] nærmere i kapitel 6.1.

2.3 Automatisk programspecialisator til Java

Der eksisterer endnu kun få partielle evaluatorer til objektorienterede sprog, alle med mere eller mindre forskellige teknikker og indsatsområder [Schultz02]. Jeg vil i dette kapitel kort beskrive JSpec [Schultz02] som er udviklet i Compose-gruppen i Rennes.

JSpec er en automatisk programspecialisator til javaprogrammer. Det vil sige at JSpec givet en programkontekst og et javaprogram, genererer programudsnit som indeholder specialiseret kode. I programkonteksten angives det hvilke bindingstider javaprogrammets instansvariabler har, hvilke værdier de ønskes specialiseret for og hvilken metode specialiseringen ønskes startet fra. Desuden tager JSpec en konfigurationsfil som indeholder konfigurationsværdier til specialiseringen. Den specialiserede kode flettes ind i den oprindelige klassefil ved at benytte aspektorienteret programmering.

Ved specialiseringen fjerner JSpec overheadet fra objektorienterede programmer producerer ved at kombinere interprocedural statisk analyse og aggressive globale optimeringer, og gennemsnitlig har JSpec givet et speed-up på 2,6 gange på et antal benchmarkprogrammer [Schultz02].

JSpec er svær at bruge for ikke-eksperter da det kræver stor indsigt i partiel evaluering og ikke mindst i JSpec at skrive programkonteksten og konfigurationensfilen. Desuden er det efter specialiseringen op til programmøren at sørge for at kalde den specialiserede metode på det rigtige tidpunkt.

Netop dét at partiel evaluering er kompliceret at bruge, afholder programmører fra at benytte specialisering til optimering af programmer hvor netop den slags optimering

kunne vise sig at være effektiv.

Rent teknisk sker der det i JSpec at javaprogrammet oversættes til C via en Java-til-C-oversætter, Harissa [Muller99], og derefter køres en partiel evaluator til C, Tempo [Consel96], som specialiserer programmet, og derefter transformeres koden tilbage til Java via Assirah, og til slut indsættes det specialiserede kode i det oprindelige program med AspectJ som jeg senere vil beskrive [Schultz02].

2.4 Deklarative og domænespecifikke sprog

Deklarative og domænespecifikke sprog er en vigtig del af dette speciale, og de følgende to kapitler vil give en generel beskrivelse af de to sprogformer.

Deklarative sprog

Når et problem opstår er det givet at man forsøger at finde den bedste løsning på problemet. Men hvordan definerer man den *bedste* løsning? En måde er at opstille nogle krav, og jo bedre disse krav er opfyldt, desto bedre er løsningen. Følgende er tre krav til at finde den bedste løsning:

- Beskrivelsen af løsningen ligger så tæt som mulig på den originale problemformulering.
- Beskrivelsen af løsningen er klar, intuitiv og præcis.
- Løsningen beskrives i termer af andre løste (del)problemer.

Disse tre er netop vigtige punkter i deklarative sprog da de beskriver deklarative sprogs virkemåde [Navarro94].

Et deklarativt sprog er et programmeringssprog der erklærer tilstande eller forhold mellem variabler, det være sig „faste variabler," det vil sige variabler som ikke ændrer sig, eller variabler i et tilhørende imperativt sprog. Disse tilstande skal være en del af løsningen på problemområdet. Jo bedre et deklarativt sprog er, des mere præcist erklæres tilstanden, eventuelt ved at bruge andre delproblemer.

Der findes ikke en entydig definition af hvad et deklarativt sprog præcist er – det afhænger af sammenhængen. Hvis man slår op i en datalogisk encyklopædi, vil man eksempelvis finde følgende: *Se funktionel programmering, logisk programmering: sprog og Setl*[2] [Ralston00]. Under hver af disse indgange bliver henholdsvist funktionel og logisk programmering beskrevet. Altså underforstået at logisk og funktionel programmering er deklarativ programmering.

[2]Setl er et programmeringssprog der bygger på en mængdestruktur, SET-Language.

En anden definition deler deklarative sprog op i tre kategorier [Paine95]:

Funktionelle Specifikationen er givet som en række funktiondefinitioner. Eksempler: Lisp [McCarthy62] og SML [Milner97].

Equational Specifikationen er givet som en række ligninger. Eksempel: OBJ3 som er et algebraisk programmerings- og specifikationssprog [Goguen88].

Logiske Specifikationen er givet som er række af prædikatdefinitioner. Eksempel: Prolog [Deransart87].

Alle de ovenstående definitioner, eller rettere forklaringer, af deklarative sprog kan bruges til at beskrive deklarative sprog, alt efter hvilket sprog, og hvilket problemområde, der bruges. Det essentielle i deklarative sprog er at man erklærer *hvad* der skal gøres i stedet for *hvordan* det skal gøres hvilket er modsat imperative sprog.

I imperative sprog som C [Kernighan88], Pascal [Jensen75] og Trine [Schmidt95] og objektorienterede sprog som Java [Flanagan97] og Beta [Madsen93] specificeres en sekvens af udtryk der skal beregnes i denne rækkefølge og som giver et resultat – man beskriver altså hvordan resultatet skal beregnes. For deklarative sprog er rækkefølgen af erklæringer ikke afgørende.

Som beskrevet ligger deklarative sprog tæt på deres problemområde hvilket gør dem meget udtryksfulde, og det er deres styrke. Et program skrevet i et imperativt sprog bliver let uoverskueligt blot det fylder mere end et par hundrede linjer. Fordelen ved deklarative sprog er at de ikke (nødvendigvis) bliver mere uoverskuelige jo mere der tilføjes da de ofte bygger på prædikater og erklæringer med oprindelse i matematik, og den logiske struktur underbygger problemløsningen [Navarro94].

Domænespecifikke sprog

Et programmeringssprog der gennem en passende notation og abstraktion er udtryksfuldt inden for et, muligvis begrænset, problemområde kaldes domænespecifikt. Desuden er domænespecifikke sprog ofte også deklarative [Deursen00], og dermed har de to meget tilfælles.

En dedikeret tilgang giver en bedre løsning til en mindre mængde problemer end en generel tilgang, og det er de domænespecifikke sprogs force. Hvor brugbart et domænespecifikt sprog er afhænger af dets *problemområde*. Når et sprog er dedikeret et bestemt problemområde, er det muligt at gøre sproget *fokuseret* på problemområdet hvilket forhåbentlig vil give sproget en styrke i forhold til tillæring og brug.

Slutbrugerprogrammering, det vil sige programmering foretaget af for eksempel brugeren af et regneark, forgår ofte i domænespecifikke eller såkaldte *script*-sprog som er i

familie med domænespecifikke sprog.

Som for deklarative sprog gælder det for domænespecifikke sprog at en af deres fordele er at de er præcise, ofte selvdokumenterende, og kan genbruges til flere formål [Deursen00].

2.5 Aspektorienteret programmering

Når man programmerer i et sprog der fordrer modularitet som for eksempel et objektorienteret sprog, ender det tit med at programmet indeholder funktionalitet der bruges i mange forskellige objekter, men på grund af modulariteten er det ikke muligt at genbruge denne funktionalitet uden at bryde objektmodelstrukturen. For eksempel har fejlhåndtering en tendens til at sprede sig ud i hele systemet. Når funktionalitet spredes på den måde, siges det at *tværsnitte*[3] (to cross-cut). Kode der tværsnitter er svær at vedligeholde da den er fordelt ud på mange objekter og derved kan være svær at finde [Kiczales97, Kiczales01].

En løsning på problemet med at vedligeholde tværsnittende metoder er at lade metoden være defineret et sted og tillade at den tværsnitter objekter uafhængig af modellen. Denne „nye“ programmeringsform kaldes aspektorienteret programmering (AOP) og er introduceret af Kiczales [Kiczales97]. Et *aspekt* er altså et modul der tværsnitter programmet. Ved at benytte aspekter bliver koden lettere at udvikle og vedligeholde, og aspektorienteret programmering kan forbedre programmets effektivitet.

AspectJ er et aspektorienteret programmeringsprog som bruges til javaprogrammer. I AspectJ er det muligt at introducere nye metoder og instansvariabler i klassefiler eller at tilføje kode før, efter eller rundt om et givet programpunkt. De introducerede programdele kan derefter kaldes som var de en del af det oprindelige program, og de programdele der er ændret for eksempel før en givet metode vil da blive kaldt når metoden kaldes. For at finde et bestemt programpunkt bruges udtrykket *pointcut*. Et pointcut kan findes ved at opskrive et slags regulært udtryk for eksempelvis en bestemt metodegruppe. Et lille illustrerende eksempel på AspectJs funktionalitet er et objekt som modellerer et punkt med en x- og en y-koordinat. Punktklassen har en set-metode for hver af punkterne. Vi ønsker nu at hver gang koordinaterne til et punkt sættes, skal de sættes til at være 10 større end den værdi set kaldes med. I stedet for at ændre i alle set-metoderne kan man opskrive et pointcut med et regulært udtryk som finder alle metoder der hedder noget med „set“ og tager et heltal som argument. Derefter kan man ændre heltallet til den ønskede værdi og forsætte kaldet af set-metoden med den

[3] *cross cut* har på engelsk mange betydninger. Den „oprindelige“ betydning er genvej, men jeg synes ikke dette ord fungerer ikke i denne sammenhæng på dansk. *Cross* kan også oversættes med modvirke eller modarbejde hvilket jeg gerne vil have ind i begrebet på dansk ved at bruge tvær(s).

nye værdi. Den nye kode skal kun skrives én gang for alle set-metoderne og kan let ændres. Eksemplet her er meget lille og måske ikke videre brugbart, men bør give en indsigt i hvad det er AspectJ kan bruges til.

2.6 Sammenfatning

I dette kapitel begyndte jeg med at gennemgå partiel evaluering og de metoder der benyttes til partiel evaluering. Jeg beskrev hvordan partiel evaluering kan være svært at bruge for ikke-eksperter hvilket er med til at partiel evaluering ikke har så stor en udbredelse på trods af den åbenlyse gode ide, nemlig at bruge det vi allerede ved om et program uden at gøre programmet „ensporet“ så det kun passer til en bestemt brug. Desuden beskrev jeg hvordan partiel evaluering er specielt effektivt ved objektorienterede programmer hvor man ofte programmerer med henblik på en model af et problemområde nærmere end med henblik på effektivitet. Dermed har jeg motiveret at gøre partiel evaluering lettere at bruge og min løsning er at definere et domænespecifikt og deklarativt sprog hvor domænet netop er partiel evaluering. For at give forståelse for hvad det er for en type sprog resten af specialet omhandler, gjorde jeg i kapitel 2.4 rede for hvad der forstås ved domænespecifikt og deklarativt. Endelig beskrev jeg i kapitel 2.5 en form for programmering der strider lidt mod de objektorienterede principper jeg ellers er opdraget til at tænke i, nemlig aspektorienteret programmering. Denne programmeringsmetode er uundværlig i specialiseringen af objektorienterede programmer hvor jeg netop ønsker at ændre på programmer der allerede er skrevet – præcis hvordan metoden bruges vil fremgå senere i specialet.

3 Pesto

Dette speciale har til formål at give ikke-eksperter en metode til at benytte partiel evaluering af objektorienterede sprog, i dette tilfælde ved at bruge JSpec til specialisering af javaprogrammer. JSpec skal bruge en kompliceret programkontekst og det er blandt andet den kontekst programmøren skal slippe for at skrive. Konteksten skal altså kunne genereres automatisk. Men det er stadig nødvendigt at kunne erklære hvad det er for nogle værdier et program skal specialiseres med hensyn til, for eksempel at eksponenten *exp* i eksemplet fra kapitel 2.1 skal være 3. Det der er brug for, er altså et simpelt deklarativt sprog hvor de specialiserede værdier kan angives. Da ideen er at det ikke skal være kompliceret at bruge skal den deklarative notation gerne ligne noget javaprogrammøren allerede kender. Det deklarative sprog jeg har udviklet hedder Pesto, *Partiel Evaluering – Specialisering Til Objektorienterede programmer*. Til at oversætte Pesto til noget JSpec forstår, har jeg skrevet en oversætter.

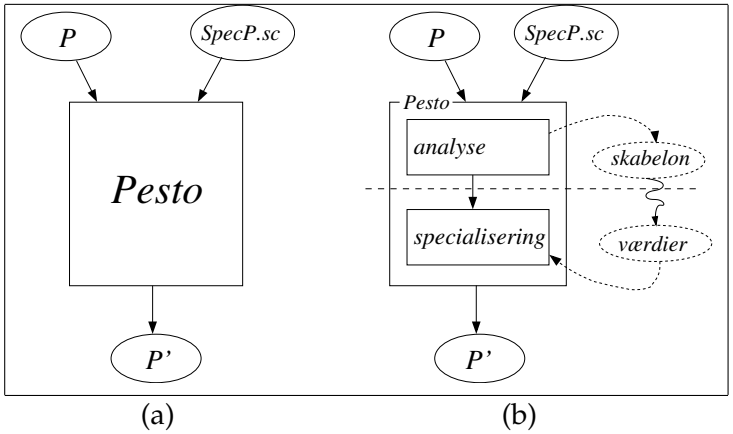
I de følgende kapitler vil jeg give en generel beskrivelse af sproget Pesto og give nogle eksempler på hvordan det bruges. Kapitel 3.1 giver en kort forklaring af det miljø Pesto-oversætteren er en del af, og resten af kapitel 3 har til formål at give læseren en god forståelse for hvad Pesto er, og hvad det kan bruges til; inden jeg i de følgende kapitler går i dybden og forklarer hvordan sproget og systemet er opbygget.

Det er begrænset hvilken funktionalitet der kan puttes ind i eksponentialfunktionen, og for at give et indblik i hvad Pesto ellers kan udtrykke vil jeg komme med et lidt større eksempel. Eksemplet vil blive beskrevet i kapitel 3.4, og efterfølgende vil specialiseringen af eksemplet blive beskrevet.

3.1 Kort fortalt

Det deklarative sprog Pesto er bygget op af specialiseringsklasser der indeholder prædikater der erklærer hvad der skal specialiseres. En specialiseringsklasse er knyttet til en javaklasse. Specialiseringsklasserne skal ligge i det filkatalog hvor det program der skal specialiseres ligger.

Pesto-oversætteren indlæser en specialiseringskontekst, *SpecP.sc*, skrevet i Pesto og pro-



Figur 3.1: Pesto-oversætterens grænseflade.

ducerer ved at bruge det program P der ønskes specialiseret et specialiseret programudsnit. Det specialiserede programudsnit indsættes i det oprindelige program, og Pesto indsætter en *vogter* der sørger for at kalde den specialiserede kode når betingelserne for at udføre koden er opfyldt. Jeg har tidligere beskrevet at programspecialisering kan bruges til optimering af plads og tid. Hvis et program skal specialiseres for at optage mindre plads er det underforstået at det nye program kun kan køres med en programkontekst der opfylder de specialiserede værdier og det specialiserede kode overskriver det oprindelige kode. I resten af specialet vil jeg ikke tage hensyn til denne form for specialisering, men kun beskæftige mig med programmer der indeholder *både* den oprindelige kode *og* den specialiserede kode, og det burde være klart at et sådan program fylder mere en et der kun indeholder den oprindelige kode. Men da kode netop ofte bliver reduceret når det specialiseres er det en vigtig sidegenvist som kan udnyttes til at skrive programmer til små specialiserede computere.

En specialiseringskontekst skrevet i Pesto er en specialiseringsklassefil med en eller flere *specialiseringsklasser* som alle er relaterede til det javaprogram de kaldes med. Figur 3.1 illustrerer processen. Selve oversætteren er tegnet som en „sort æske“ – jeg vil senere uddybe præcis hvad der foregår i Pesto. På figuren har jeg ladet Pesto indlæse både konteksten og programmet på trods af at Pesto kun tager et argument, nemlig specialiseringsklassefilen. Programmet P er nødvendigt for Pesto, men Pesto bruger det ved at læse de filer der hører til P direkte i det filkatalog hvor det ligger, og for at illustrere dette har jeg på figuren vist at P bliver indlæst af Pesto. Figuren viser to scenarier; et simpelt (a) hvor Pesto indlæser filer og spytter et javaprogram P' ud. Det andet scenario (b) kræver interaktion med brugeren. Oversætteren udfører analysen og producerer en skabelon som brugeren skal ændre i så den indeholder de værdier

```
public class Power {
  int exp;

  public static void main(String[] args) {
    Power p = new Power(2);
    System.out.println(p.raise(2));
    p = new Power(3);
    System.out.println(p.raise(2));
  }

  public int raise(int base) {
    int result = 1;
    for (int i = 0; i < exp; i++) {
      result = result * base;
    }
    return result;
  }

  public Power(int exp) {
    this.exp = exp;
  }

  public Power() {
  }
}
```

(a)

```
specclass Cube specializes Power {
  exp == 3;
  public int raise(int base);
}
```

main file: *Power.java*

(b)

Figur 3.2: Javakoden til eksponentialfunktionen, samt specialiseringsklasse til specialisering af *exp*.

P skal specialiseres med hensyn til. Hvad denne fil præcist indholder og hvordan den skal ændres, vil blive beskrevet senere i dette kapitel. Når brugeren har givet oplysninger om de faktiske værdier, kan Pesto køre videre og selve specialiseringen vil finde sted, altså det der på figuren ses under den punkterede linje. Også her er produktet af specialiseringen naturligvis et javaprogram. I begge tilfælde gælder der at P og P' er semantisk ækvivalente – det vil sige at de med samme programkontekst producerer samme resultat. Forskellen er blot hvordan resultatet bliver beregnet.

Koden i figur 3.2 (a) er en objektorienteret version af eksponentialfunktionen fra kapitel 2.1 skrevet i Java. Eksponentialfunktionen raise kaldes ved at instansiere et Power-objekt med en eksponent og derefter kalde metoden raise på objektet. Bemærk at klassen Power har en *default constructor* hvilket af tekniske årsager er at krav til de klasser der skal specialiseres; et andet krav er at de instansvariabler der skal specialiseres

```
public aspect Cube{
  static private int Power._raise_spec(int vi1) {
    int si0;
    si0 = 1 * vi1 * vi1 * vi1;
    return si0;
  }
  private boolean Power._guard_SpecPower(int base){
    if (!(this.exp == 3)) return false;
    return true;
  }
  pointcut _raise_entrypoint (Power _power, int _raise_base):
  call (int Power.raise(int))
    && args(_raise_base)
    && target(_power);
  int around(Power _power, int _raise_base):
  _raise_entrypoint(_power, _raise_base) {
    if (_power._guard_SpecPower(_raise_base)) {
      return _power._raise_spec(_raise_base);
    } else {
      return proceed(_power, _raise_base);
    }
  }
}
```

Figur 3.3: Kode genereret af Pesto og JSpec.

skal være *public*. Følgende vil beregne 2^3 og resultatet 8 vil være i variablen *result*:

```
Power p = new Power(3);
int result = p.raise(2);
```
(4)

Figur 3.2 (b) viser specialiseringsklassen Cube til Power. Cube har to erklæringer; den første er et *prædikat* der udtrykker at Power skal specialiseres for *exp* = 3 og den anden erklærer *hvor* specialiseringen skal starte, nemlig i metoden raise. Et sådan *indgangspunkt* (*entry point*) er nødvendigt i alle specialiseringer for at vide hvilket udsnit af programmet der skal specialiseres. Til gengæld må der kun være ét indgangspunkt i en specialisering. Man kan sagtens forestille sig partiel evaluering med flere indgangspunkter, men det understøtter JSpec, og dermed Pesto, ikke. I eksemplet i figur 3.2 er der kun én specialiseringsklasse, men senere vil jeg vise eksempler med flere specialiseringsklasser til én specialisering. Alle specialiseringsklasser til en specialisering skal ligge i den samme *specialiseringsklassefil*. I filen skal det være angivet hvor programmets *main*-metode er hvilket gøres i bunden af specialiseringsklassen Cube med linjen *main file*: *Power.java* .

	Tid		
Virtuel maskine	**Oprindeligt program**	**Specialiseret program**	**Forbedring**
Hotspot 1.3	3718	3832	0%
-server	1058	716	48%
Hotspot 1.4	9838	2720	263%
-server	1259	706	77%
IBM JIT 1.3.1	1115	783	42%

Figur 3.4: Forbedringen ved den specialiserede eksponentialfunktion.

Figur 3.3 viser den kode Pesto ved hjælp af JSpec har genereret ud fra specialiseringsklassen `Cube` og javaprogrammet `Power`. Koden er et *aspekt* som AspectJ sætter ind i den oprindelige `Power`-klasse. Metoden `_raise_spec` er den specialiserede metode som JSpec har genereret (at der står „`Power.`“ foran metoden, er AspectJ-terminologi og betyder blot at metoden skal indsættes i klassen `Power`). Som det ses, er koden ækvivalent med den specialiserede kode i kodeeksempel 2 fra kapitel 2.1. Ud over den specialiserede metode indeholder aspektet en vogter genereret af Pesto. Selve vogteren er den metode der hedder `_guard_SpecPower`. Vogteren sørger for at den specialiserede kode bliver kaldt når *exp* er tre. Resten af aspektet sørger for at finde det rigtige sted at putte koden ind. Koden vil blive nærmere gennemgået i kapitel 5.4.

Eksperimenter

For at vise forskellen på det oprindelige kode og det specialiserede har jeg kørt de to programmer og taget tid på hvor meget hurtigere det specialiserede kode kører. Det specialiserede kode er genereret af JSpec, og konteksten til JSpec samt vogteren er genereret af Pesto. Vogteren er placeret mest hensynsfuldt hvilket vil sige uden for det kritiske kode hvilket jeg vil komme nærmere ind på senere.

For at få det mest objektive og sikre resultat har jeg kørt programmet et givet antal omgange for at få stabile tider (i dette tilfælde 50 mio. iterationer) og hver gang ladet basis være resultatet af den forrige kørsel modulo 4. Hver sådan omgang har jeg kørt 10 gange, og tidsresultatet er gennemsnittet af de sidste fem kørsler for at være sikker på at det er det optimerede kode jeg måler på og ikke cache-effekt. Alle eksperimenter i dette speciale er kørt efter denne skabelon på en Dual Pentium III CPU 1 GHz maskine med 16 Kb instruktions-cache og 16 Kb data-cache på level 1-cache og 256 Kb level 2-cache på hver af de to processorer. Desuden har maskinen 1 Gb hukommelse og kører Linux 2.4.18-19.7.

Der er stor forskel på virtuelle maskiner til Java og deres optimeringsstrategier, og derfor har jeg valgt at køre programmerne med forskellige virtuelle maskiner hvilket giver meget forskellige resultater. Tabellen i figur 3.4 viser forbedringen ved de forskellige virtuelle maskiner. Hotspot har en parameter, *server*, der udløser ekstra optimeringer, men som det ses i tabellen er det ikke nødvendigvis med de ekstra optimeringer at der er størst fordel ved at specialisere.

Anden kolonne viser udførselstiden i på det oprindelige kode, og denne kolonne er med for at vise forskellen på de forskellige virtuelle maskiner og sætte forbedringerne ind i en sammenhæng. Alle tider i eksperimenterne måles i millisekunder. Tredje kolonne er udførselstiden af det optimerede program. Kolonne fire er den procentvise forbedring af udførselstiden, og som det ses er den forbedring der ser bedst ud ikke nødvendigvis den der udfører det specialiserede kode hurtigst. Hotspot 1.3 -server er den hurtigste på det oprindelige kode, og det bedste specialiseringsresultat giver Hotspot 1.4 hvor programmet kører 2,6 gange hurtigere ved at benytte det specialiserede kode. Det viser sig dog at Hotspot 1.3 -server, Hotspot 1.4 -server og IBMs oversætter kører næsten lige hurtigt med den specialiserede kode.

Dette kapitel var en kort gennemgang af hvordan Pesto og havde til formål at give læseren overblik. Resten af dette kapitel er en mere detaljeret gennemgang af Pesto.

3.2 Specialiseringsklasser

Man kan opfatte forholdet mellem en javaklasse og tilhørende specialiseringsklasse som en slags nedarvning hvor specialiseringsklassen arver alt fra javaklassen og tilordner nogle specifikke værdier til javaklassens instansvariabler. Specialiseringsklassen kan dog ikke tilføje ny funktionalitet til javaklassen hvilket et en vigtig pointe. De to klasser skal være semantisk ækvivalente [Volanschi97].

En specialiseringsklasse er en *kvasi-invariant* [Cowan96] for programmet. En invariant for et program, eller som oftest et programudsnit, er et udsagn der altid er sandt på det programpunkt hvor invarianten hører til. En kvasi-invariant er som oftest sand, men ikke altid. I konteksten af specialiseringsklasser gælder det, at *når* en kvasi-invariant er sand, kan programmet anvende specialiseret kode.

Som det fremgår af specialiseringsklassen i figur 3.2 (b), skal man specificere hvilke javaklasser man ønsker at specialisere. I `Cube` drejer det sig om `Power`. `Cube` har desuden et indgangspunkt, og under specialiseringen er det det kode der kan nås fra indgangspunktet der specialiseres.

En specialiseringsklasse består af et antal prædikater som til sammen giver klassens kvasi-invariant. Hvis en fil består af flere specialiseringsklasser, skal de være forbundet

så de alle kan nås gennem den specialiseringsklasse der indholder indgangspunktet; de specialiseringsklasser der ikke indholder indgangspunktet skal der dermed være direkte eller indirekte *referencer* til fra specialiseringsklassen med indgangspunktet. Hvordan man refererer mellem specialiseringsklasser, vil jeg beskrive i detaljer i kapitel 4.7.

3.3 Bindingstidstegn

Som nævnt i kapitel 3.1 er der en mere avanceret brug af Pesto end den ovenfor nævnte. I kapitel 2.1 beskrev jeg hvordan den første fase i specialiseringen er bindingstidsanalysen hvor variablerne deles op i statiske og dynamiske. Under analysen er det ikke afgørende hvilke faktiske værdier de statiske værdier har. Det er først under selve specialiseringen at de faktiske værdier benyttes. Når jeg taler om bindingstid, er *dynamisk* og *statisk* de to vigtigste begreber, hvor dynamisk som beskrevet er når værdien er ukendt, og statisk er når en variabel er kendt, *uanset* om den faktiske værdi er kendt eller ej. Hvis blot vi ved at værdien er kendt betegnes variablen som statisk.

Denne „egenskab“ ved partiel evaluering kan bruges til at undlade at give de faktiske værdier før efter analysen. Analysen vil på den måde kunne bruges til flere forskellige faktiske værdier for den samme statiske variabel. I eksemplet i figur 3.2 skal det altså være muligt at udtrykke at *exp* er statisk og så siden, efter analysen, give den faktiske værdi.

I Pesto arbejder jeg med tre begreber der knytter sig til bindingstider: *dynamisk*, *implicit statisk* og *eksplicit statisk*. Dynamisk bruges på samme måde som ved bindingstider, altså at værdien er ukendt. Alle instansvariabler er dynamiske for specialisatoren indtil andet er erklæret. Det der for bindingstidsanalyse betegnes som statisk har jeg delt op i to begreber; implicit og eksplicit statisk. En implicit statisk instansvariabel er kendt, men den faktiske værdi kendes ikke – husk at for bindingstid er det ikke afgørende om den faktiske værdi er kendt. Eksplicit statisk er netop når værdien er kendt, *og* den faktiske værdi kendes som i eksemplet i figur 3.2 hvor *exp* siges at være eksplicit statisk. Distinktionen mellem statisk, implicit statisk og eksplicit statisk er vigtig i resten af specialet.

Det er i Pesto muligt at erklære prædikater for instansvariabler uden at benytte den eksplicit statiske værdi, nemlig ved at bruge *bindingstidstegn*. For simple udtryk bruges et spørgsmålstegn (?) til at erklære at en instansvariabel er dynamisk og et udråbstegn (!) tilkendegiver at instansvariablen er implicit statisk. For simple udtryk som det i

Cube (figur 3.2) kan programmet specialiseres mere generelt ved at bruge udråbstegnet:

```
specclass KnownExp specializes Power {
  exp == ! ;
  public int raise( int base ) ;
}
```
(5)

KnownExp indlæses af Pesto, og programmet bliver specialiseret efter formen i figur 3.1 (b) hvor analyse og specialisering foregår i to faser. Efter endt analyse skriver Pesto en fil, *templateKnownExp.values*, som med et værktøj kan modificeres så den indeholder de ønskede specialiseringsværdier. Filen har følgende indhold:

```
KnownExp {
  exp = int;
}
```
(6)

Når specialiseringen fortsætter, forventer Pesto at filen er modificeret så „int" er skiftet ud med et heltal – den værdi der ønskes specialiseret med hensyn til – og at ændringerne ligger i en fil med efternavnet *.values*. Følgende er indholdet af den modificerede fil *Cube.values* som angiver at eksponenten skal være tre:

```
KnownExp {
  exp = 3;
}
```
(7)

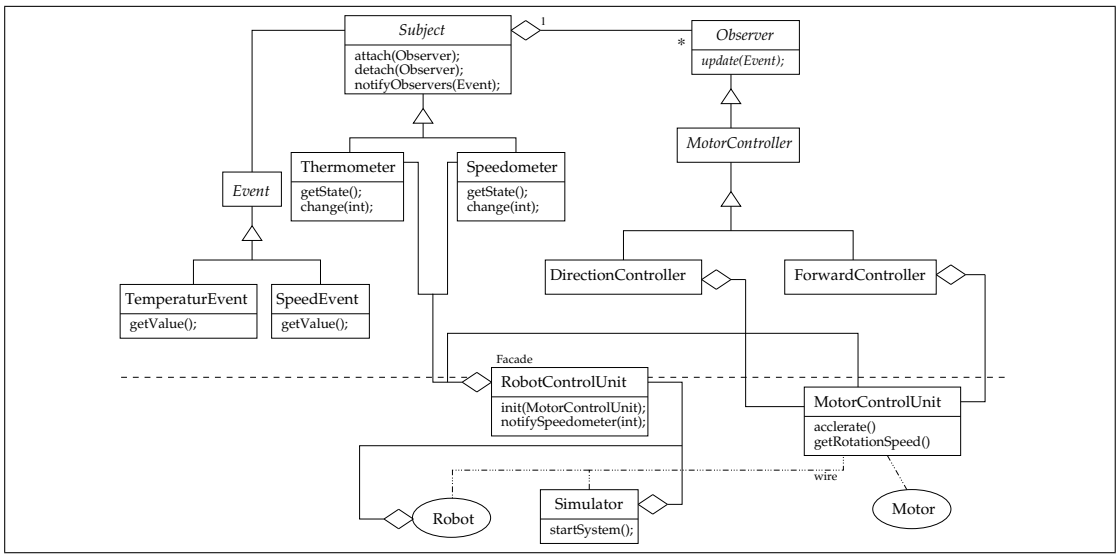
Selve ændringen kan ske ved at hente filen ind i en teksteditor og ændre i den eller ved at benytte et dertil indrettet værktøj hvor det kun er muligt at fortage lovlige ændringer. *Skabelon-* og *værdifilerne* vil blive beskrevet i kapitel 4.6, og funktionaliteten af et muligt værktøj vil blive diskuteret i kapitel 6.2 om perspektiverne for Pesto.

3.4 Et eksempel

For at give et bedre indblik i Pestos formåen vil jeg i dette kapitel beskrive et lidt større eksempel som i de følgende kapitler vil blive brugt til at illustrere hvordan specialisering kan bruges i realistiske problemområder.

Objektorienteret programmering har lige fra de første objektorienterede sprog været yderst velegnet til simulation. Faktisk var Simula, som betegnes som det første objektorienterede sprog, netop designet til at simulere virkeligheden. Fordelen ved at benytte objektorientering til simulation er som tidligere beskrevet at man modulerer sit problemområde med generelle komponenter der sammensættes og derved giver en afbildning af det virkelige problem. Det giver overblik, men desværre også overhead som beskrevet i kapitel 2.2. Dette overhead kan nedbringes ved at bruge partiel evaluering hvilket dermed gør partiel evaluering yderst velegnet til simulering.

Eksemplet i dette kapitel er en Legorobot. Robotten kan programmeres, men fordi den



Figur 3.5: Klassediagram af roboteksemplet. Klasser er repræsenteret som kasser og fysiske objekter som ovaler.

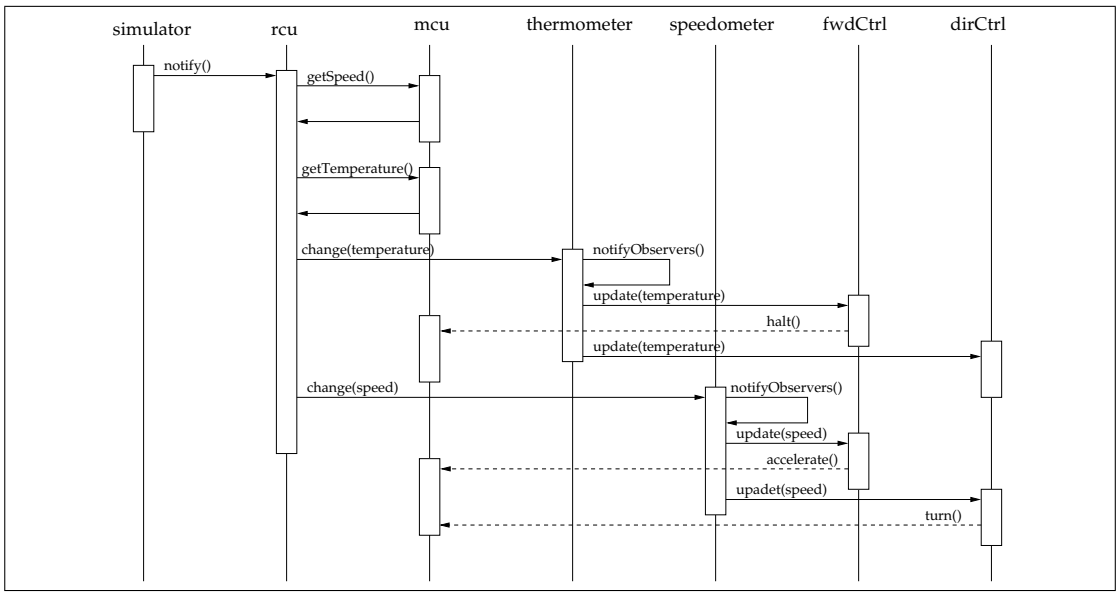
har begrænset processorkraft, er der ikke meget plads til programmet. Specielt ikke da dette ønskes at være objektorienteret og endnu værre, der ønskes at bruge et *designmønster*. Derfor vil jeg anvende specialisering på det program der styrer robotten.

Robotten kan køre frem og den kan dreje – den har en hastighed og en retning som er afhængig af robottens motors temperatur. For at modellere dette er det oplagt at benytte designmønstret „Observer“.

„Observer“ er desuden interessant i forhold til specialisering da det er et ofte brugt mønster. Designmønstret ses i bilag A.

Kort fortalt består mønstret af en eller flere *subjekter* (*subjects*) som overvåges af et antal *observatører* (*observers*). Hvis der sker ændringer i et subjekt, gives der besked til observatørerne der så muligvis opdateres. Observatørerne kender ikke noget til hinanden. Observer-mønstret kaldes også for *Publisher-Subscriber*.

Da jeg ikke har en robot at afprøve programmet på, simuleres robotten. Figur 3.5 viser klassediagrammet for robotsystemet. Klassen Simulator simulerer den fysiske robot, og simulatoren har en RobotControlUnit som er robottens styreenhed, og desuden er en facade til resten af systemet. Det vil sige at simulatoren kun bekymrer sig om RobotControlUnit og kun kan bruge metoder i denne klasse. Facaden repræsenteres i figuren af den punkterede linje. Som det ses, er det ikke alt andet der ligger bag facaden; den enhed der skal styre motoren, MotorControlUnit, ligger også uden for facaden. Simulatoren



Figur 3.6: Interaktionsdiagram for roboteksemplet.

bruger oplysninger fra denne til at bestemme robottens hastighed, og derfor ligger den også uden for facaden.

Som beskrevet ovenfor skal motoren reagere behørigt på temperatur- og hastighedsændringer. Temperaturen og hastigheden er *subjekter* der skal *observeres*. Som det ses i diagrammet er Thermometer og Speedometer subklasser af Subject (se beskrivelsen og diagrammet i bilag A). De ændringer der rapporteres i speedometeret og termometeret videresendes til motoren af to styreenheder, ForwardController og DirectionController. Disse styreenheder er subklasser af den abstrakte klasse *MotorController* som igen er en subklasse af *Observer*. Hver af de to subjekter, Thermometer og Speedometer, har en reference til hver af de to styreenheder der er mønstrets *observatører*.

Arbejdsgangen

Jeg vil i dette kapitel give et eksempel på en *arbejdsgang* i systemet, det vil sige de metodekald der foretages for at udføre en enkelt „runde“ i systemet, altså fra simulatoren henter oplysninger fra motoren til den gør det igen. Først initialiseres systemet ved at RobotControlUnit initialiseres fra Simulator. Alle klasser oprettes, observatører, det vil sige kontrolenheder til hastighed og retning, tilknyttes og motoren starter med hastighed 0. Til at begynde med initialiseres speedometeret til en hastighed på -1. Figur 3.6 viser en „runde“ i motoren. En løkke i simulatoren henter gennem RobotControlUnit (*rcu*) fra MotorControlUnit (*mcu*) den hastighed motoren kører med, og motorens temperatur.

Kontrolenheden *rcu* sender en besked til speedometeret og termometeret om de aktuelle værdier. Hvis der er en ændring i hastigheden[1] eller temperaturen, sendes denne til motorstyreenhederne ved at benytte Observer-mønstrets metode notifyObservers( ) som sender ændringerne til systemets observatører, *fwdCtrl* og *dirCtrl*, via metoden update. Hastighedsenheden *fwdCtrl* reagerer på update ved eventuelt at bremse motoren hvis motoren er ved at blive for varm, eller hvis hastigheden er for lav og temperaturen er fin, ved at accelerere motoren. Denne arbejdsgang fortsætter til motoren kører med 20 omdrejninger i sekundet. Når motoren kører over 18 omdrejninger i sekundet stiger temperaturen i motoren. Når temperaturen når 28 grader sættes hastigheden ned og når motoren er kølet lidt af, startes forfra med at nå hastigheden 20. Hvis hastigheden er mellem 18 og 20 drejer lader retningsenheden *dirCtrl* robotten dreje lidt til venstre og hvis hastigheden er mellem 20 og 22 drejer den lidt højre.

Omkostninger ved brug af Observer

Ved at bruge designmønstret får man fordele såsom modularitet og let vedligeholdelse. Men der er omkostninger ved at bruge designmønstre, som beskrevet i kapitel 2.2. I roboteksemplet er der lang vej fra aflæsningen af hastigheden til at hastigheden ændres hvilket kan illustreres ved i figur 3.5 at følge vejen fra robotstyreenheden til update i motorenheden. Hvis systemet var skrevet i hånden uden brug af designmønstre, ville man måske have lavet et direkte kald fra RobotControlUnit til MotorControlUnit. Her kan der sættes ind med specialisering. Det er ikke muligt at specialisere eksemplet til at ligne håndskrevet kode, men det er godt at have som ideal.

Desuden er der de dyre virtuelle kald til observatørerne som man i håndskrevet kode ville udskifte med direkte kald hvilket også er et oplagt sted at sætte ind med specialisering.

3.5 Specialisering af eksemplet

For at specialisere eksemplet vil jeg i dette kapitel benytte nogle konstruktioner i Pesto som jeg endnu ikke har forklaret. Når de dukker op vil jeg kort forklare og derefter vil jeg bede læseren være tålmodig – mere detaljeret beskrivelse følger. Jeg ønsker blot at bruge dette eksempel til at vise hvad Pesto formår inden jeg går i gang med den mere detaljerede beskrivelse.

Som nævnt i forrige kapitel er det vejen fra RobotControlUnit til MotorControlUnit jeg gerne vil gøre mindre omstændeligt. Specialiseringsklasserne til en specialisering af eksemp-

[1] En ændring i hastigheden vil her sige at der er forskel på den værdi speedometeret viser og på den aktuelle hastighed hentet fra motorenheden. Denne forskelsberegning er også grunden til at speedometeret initialiseres til -1 da en ændring ellers aldrig ville opdages.

```
specclass MyRobot specializes Simulator {
  rcu : KnownRobot;

  public int startSystem( int o ) ;
}
specclass KnownRobot specializes RobotControlUnit {
  thermometer : SpecThermo;
  speedometer : SpecSpeedo;
}
specclass SpecThermo specializes Thermometer {
  observers : Observer[2] = {SpecFwdCtrl, DirectionController};
}
specclass SpecSpeedo specializes Speedometer {
  observers : Observer[2] = {SpecFwdCtrl, DirectionController};
}
specclass SpecFwdCtrl specializes ForwardController {
  topTemperature == 28;
  goodSpeed == 20;
}
main file : Main.java
extra classes : {Event, SpeedEvent, TemperatureEvent}
```

Figur 3.7: Specialiseringsklasser til specialisering af acceleration af robotmotoren.

let ses i figur 3.7. Det jeg ønsker at specialisere er RobotControlUnit hvor jeg kender de præcise typer på subjekterne, og ydermere kender jeg observatørerne i subjekterne og ved i hvilken rækkefølge de står. Kaldet fra RobotControlUnit starter i metoden notify, men denne metode er et dårligt indgangspunkt da metoden bliver kaldt ved hver iteration af motoren; og da betingelserne for specialiseringen bliver kontrolleret når et indgangspunkt kaldes, ville det give et overhead. I stedet vælger jeg den metode der starter robotten, nemlig startSystem i Simulator. Jeg skal blot være sikker på at de værdier jeg specialiserer for rent faktisk er tilordnet på det tidspunkt i programmet, men denne betingelse er opfyldt da hele systemet initialiseres når Simulator initialiseres.

I Simulator kan jeg specialisere *rcu* med hensyn til at instansvariablen har typen RobotControlUnit ved at erklære følgende:

rcu: RobotControlUnit; (8)

Dette er dog ikke en særlig interessant specialisering da oversætteren allerede kender typen af *rcu*. Og det jeg i virkeligheden er interesseret i at specialisere er RobotControlUnit. Specialiseringsklassen KnownRobot specialiserer netop RobotControlUnit.

For at KnownRobot kan bruges af specialiseringen skal den kaldes fra MyRobot. Det gøres ved at specialisere med hensyn til at *rcu* har typen KnownRobot:

rcu : KnownRobot; (9)

Men der er ting i klassen Thermometer jeg også gerne vil specialisere for, nemlig at jeg kender subjektets observatører, og derfor har Thermometer og Speedometer hver deres specialiseringsklasse, SpecThermo og SpecSpeedo. På samme måde skal MyRobot direkte eller indirekte referere til disse specialiseringsklasser, og det gøres gennem KnownRobot.

Observatørerne opbevares i subjekterne i et array af typen Observer[ ] i instansvariablen *observers*. Længden af *observers* er lig antallet af observatører. Følgende linje i SpecThermo specialiserer for at *observers* har længden to og at de to elementer er henholdsvis af typen SpecFwdCtrl og DirectionController i den rækkefølge. SpecFwdCtrl refererer igen til en specialiseringsklasse der specialiserer ForwardController:

observers: Observer[2] = {SpecFwdCtrl, DirectionController}; (10)

Det var selve specialiseringen af eksemplet. Men for at Pesto kan oversætte specialiseringsklasserne til en kontekst som JSpec forstår, er der brug for endnu nogle informationer om det program der skal specialiseres. I bunden af specialiseringsklassefilen står følgende linjer:

main file : *Main.java*
extra classes : {Event, SpeedEvent, TemperatureEvent} (11)

Programmets main-metode ligger i filen *Main.java*, og *extra classes* er de klasser der skal bruges i specialiseringen, men som der ikke refereres til i specialiseringskonteksten. De ekstra klasser er ikke obligatoriske i Pesto da de ikke altid er nødvendige. Men hvis det er nødvendigt og klasserne ikke er specificeret, kan programmet ikke blive specialiseret, og der vil blive givet en fejl. Pesto kan ikke kontrollere om specialiseringen bruger flere klasser end de der allerede er specificeret i specialiseringsklasserne da Pesto ikke kender noget til programmets struktur. Fejlen kommer derfor først når selve specialiseringen er i gang, det vil sige sent i processen. Der er derfor afgørende at programmøren kender sit program og kan erklære de ekstra klasser.

3.6 Hvad sker der i det specialiserede kode

Under programspecialiseringen af programmet bruger JSpec de oplysninger der er givet i specialiseringsklasserne til at optimere koden.

Rent praktisk er der to interessante optimeringer forårsaget af specialiseringsklasserne; de påvirker metoderne notifyObservers i klassen Subject, som ses i figur 3.8, og update i klasserne ForwardController og DirectionController. Figur 3.9 viser update-metoden for

```
public void notifyObservers(Event e) {
  for (int i = 0; i < observers.length; i++) {
    ((Observer)observers[i]).update(e);
  }
}
```

Figur 3.8: Den oprindelige metode notifyObservers i klassen Subject.

ForwardController. Metoden notifyObservers består af en løkke der løber så mange gange som længden af *observers*-arrayet og for hver observatør kalder dennes update-metode. Men med specialiseringsklasserne er længden af arrayet kendt og det vides præcis hvilke observatører der er i arrayet. Løkken kan altså foldes ud, og der kan kaldes direkte til de pågældende observatører. Figur 3.10 viser et udsnit af den specialiserede kode og som det ses er notifyObservers blevet til to forskellige metoder med næsten identisk indhold. Forskellen på metoderne er at de bliver kaldt med forskellige subklasser af Event, nemlig SpeedEvent og TemperatureEvent, som argument. Løkken i metoden er foldet ud og i det der før var løkkens krop, er de virtuelle kald til observatørerne elimineret ved at kalde „direkte" til de enkelte observatører hvilket gøres i JSpec ved at caste til den aktuelle type og derefter kalde direkte til de specialiserede varianter af update som er *final*. Som det ses i figuren kaldes der til fire forskellige specialiserede update-metoder; to for hver observatør med en for hver type event. Figur 3.11 viser de specialiserede update-metoder for ForwardController og som det ses, er den første metode tilegnet TemperatureEvent og den anden SpeedEvent. JSpec sørger selvfølgelig for at kalde den rigtige metode på det rigtige tidpunkt.

Eksperimenter

Det interessante er nu hvor meget specialiseringerne betyder for udførselstiden. Som i Power-eksemplet har jeg under eksperimentet kørt programmet et givet antal omgange, og en sådan kørsel gentages ti gange mens der tages tid på udførslen. Da der ikke er noget tilfældigt i koden, skal hver eneste udførsel give samme resultat. Af de fem sidste af de ti kørsler tages gennemsnittet på tiden.

Anden kolonne i tabellen i figur 3.12 viser udførselstiden for forskellige virtuelle maskiner på den oprindelige kode, og det ses at Hotspot 1.3 både er den hurtigste og den langsommeste henholdsvis med og uden -server. Tredje kolonne er udførselstiden på det samme program, men den specialiserede version af programmet.

Fjerde kolonne viser den procentvise forbedring af udførselstiden ved at bruge det specialiserede kode. I hver kørsel er der kørt 3 mio. omgange. Som det ses er der op til 71% forbedring ved at bruge den specialiserede kode med en Hotspot 1.3 oversætter, men

```
public void update(Event event) {
  if (event instanceof TemperatureEvent) {
    int temp;
    TemperatureEvent t = (TemperatureEvent)event;
    temp = t.getValue();
    if (temp > topTemperature) {
      mcu.halt();
      coolingDown = 1;
    }
    if (temp == goodSpeed) {
      coolingDown = 0;
    }
  } else {
    if (event instanceof SpeedEvent) {
      int speed;
      SpeedEvent s = (SpeedEvent)event;
      speed = s.getValue();
      if (speed < 20) {
        if (coolingDown == 0) {
          mcu.accelerate();
        }
      }
    }
  }
}
```

Figur 3.9: Den oprindelige metode update i klassen ForwardController.

faktisk er det ikke Hotspot 1.3 der har den bedste udførselstid på det specialiserede kode. Det er der imod IBMs oversætter som også har en pæn forbedring på 48%. Hvorfor de forskellige oversættere specialisere og optimere som de gør, er ikke let at give en umiddelbar forklaring på da det er sammenspillet mellem kode, specialiseringen og optimeringen der er afgørende, og der er dermed mange faktorer der spiller ind.

Hvor godt et program specialiserer afhænger meget af den kode der skal specialiseres, og det er ikke altid til at forudsige. Som det ses i tabellen, er der stor forskel på de forskellige javaoversættere. Det afhænger af hvilke optimeringer oversætteren benytter og om specialiseringerne lige præcis udløser optimeringer i en givet oversætter. Det er ikke engang givet hvilken oversætter der er bedst i forhold til specialisering. I kapitel 3.1 var det Hotspot 1.4 der gav de bedste resultater mens de bedste resultater for dette eksempel er fra Hotspot 1.3 som faktisk ingen forbedringer gav i Power-eksemplet.

At det er uforudsigeligt hvilke specialiseringer der giver øget forbedringer, har jeg også oplevet da jeg startede med at køre eksemplet uden at have funktionalitet i den ene ob-

```
final private void Subject._GnotifyObservers_1(Event e) {
   Observer[] observerArrray;
   Observer oneObserver;

   observerArrray = this.observers;
   oneObserver = (observerArrray)[0];
   ((ForwardController)oneObserver)._GForwardController_update_1_1(e);
   observerArrray = this.observers;
   oneObserver = (observerArrray)[1];
   ((DirectionController)oneObserver)._GDirectionController_update_1_2(e);
   return;
}
final private void Subject._GnotifyObservers_2(Event e) {
   Observer[] observerArrray;
   Observer oneObserver;

   observerArrray = this.observers;
   oneObserver = (observerArrray)[0];
   ((ForwardController)oneObserver)._GForwardController_update_2_1(e);
   observerArrray = this.observers;
   oneObserver = (observerArrray)[1];
   ((DirectionController)oneObserver)._GDirectionController_update_2_2(e);
   return;
}
```

Figur 3.10: Den specialiserede metode notifyObservers.

```
final private void ForwardController._GForwardController_update_1_1(Event e) {
  int s;
  MotorControlUnit aMCU;
  int s2;

  s = ((TemperatureEvent)e).getValue_1();
  s2 = s;
  if (s2 > 28) {
    aMCU = this.mcu;
    aMCU.halt_1();
    this.coolingDown = 1;
  }
  if (s2 == 20) {
    this.coolingDown = 0;
    return;
  }
  return;
}
final private void ForwardController._GForwardController_update_2_1(Event e) {
  int s;
  MotorControlUnit aMCU;

  s = ((SpeedEvent)e).getValue_1();
  if (20 > s) {
    s = this.coolingDown;
    if (s == 0) {
      aMCU = this.mcu;
      aMCU.accelerate_1();
    }
  }
  return;
}
```

Figur 3.11: De specialiserede metoder update i metoden ForwardController.

	Tid fuld version		
Virtuel maskine	**Oprindeligt program**	**Specialiseret program**	**Forbedring**
Hotspot 1.3	4425	2622	71%
-server	1222	1137	6%
Hotspot 1.4	1865	1729	8%
-server	1528	1036	47%
IBM JIT 1.3.1	1498	1012	48%
	Tid begrænset version		
Virtuel maskine	**Oprindeligt program**	**Specialiseret program**	**Forbedring**
Hotspot 1.3	3509	1869	81%
-server	1402	1011	44%
Hotspot 1.4	1626	1335	19%
-server	1459	1001	49%
IBM JIT 1.3.1	1355	1101	24%

Figur 3.12: Tider og procentvise forbedringer ved udførsel af specialiserede versioner af robotten.

servatør, DirectionController. Jeg regnede med at få bedre udførselstider hvis jeg implementerede funktionalitet i observatøren da der dermed ville være flere virtuelle kald at eliminere. Men det viste sig med nogle oversættere at være lige modsat. Nederste del af tabellen figur 3.12 viser de samme målinger som den første, men blot for den begrænsede version af programmet. Den sidste kolonne den viser den procentvise forbedring i forhold til den oprindelige kode. Det er faktisk kun med IBMs *just-in-time*-oversætter at der er en klar forbedring ved at have mere funktionalitet. Et, for mig, overraskende resultat der også blot viser at det kræver erfaring at vide hvilke specialiseringer der klare sig godt hvornår.

3.7 Sammenfatning

I dette kapitel har jeg introduceret Pesto som er mit forslag til et deklarativt sprog til specialisering af objektorienterede programmer. Sproget er bygget op af specialiseringsklasser der tilknyttes klasser i et javaprogram. Sproget er domænespecifikt til partiel evaluering – det ville ikke give mening at tale om at en instansvariable har en bestemt værdi hvis der ikke efterfølgende bliver foretaget delvis evaluering af pro-

grammet. Sproget opfylder de krav jeg tidligere stillede til deklarative sprog om at man erklærer *hvad* der skal gøres, nemlig specialiseres for en given statisk information, ikke *hvordan* det gøres. Desuden er rækkefølgen af prædikater ikke afgørende for specialiseringen. Jeg beskrev i kapitel 3.3 hvordan det er muligt at bruge *bindingstidstegn* til at angive at en instansvariabel er statisk, men at den faktiske værdi endnu ikke ønskes angivet. Denne form for statisk information benævnes *implicit statisk*, mens *eksplicit statisk* dækker de prædikater hvor en endelig værdi er givet i prædikatet. Jeg beskrev desuden kort at Pesto sørger for at den rigtige kode bliver kaldt på det rigtige tidspunkt, det vil sige at den specialiserede kode bliver kaldt når alle prædikater i specialiseringskonteksten er opfyldt og at den generiske kode i modsat fald køres. Rent praktisk indsætter Pesto en *vogter* i koden. Endelig giver jeg et detaljeret eksempel på hvordan specialisering kan bruges til „Observer“-designmønstret.

4 Design af sproget

Efter en introduktion i Pestos basale kunnen i kapitel 3 vil jeg i dette kapitel beskrive designet af Pesto. Et vigtigt krav for mig er at sproget umiddelbart skal kunne bruges til noget. I kapitel 4.1 vil jeg gøre rede for nogle af de overvejelser jeg har gjort mig under designet af sproget og under udviklingen af oversætteren. I de efterfølgende kapitler vil jeg beskrive de forskellige dele af Pesto.

4.1 Designovervejelser

Pesto er et deklarativt og domænespecifikt sprog der kan udtrykke prædikater til specialisering af javaprogrammer. Det vil sige at det er ikke muligt at lave beregninger, betingelser, løkker eller andre udtryk. Sproget indeholder kun de konstruktioner der skal bruges til specialiseringen hvilket gør sproget overskueligt.

Pesto skal bruges til specialisering af javaprogrammer, og det er derfor oplagt at syntaksen ligner Java. Det var et af kravene til Pesto før sproget blev designet. Ud over en syntaks der skulle være javalignende, var det meget vigtigt at sproget skulle være let at bruge. I forvejen har partiel evaluering ry for at være utilgængeligt, og det fremmer ikke brugen at man skal sætte sig ind et kompliceret sprog eller en kompliceret syntaks for at benytte partiel evaluering.

Når først man har gjort sig klart at sproget skal være let at bruge, er det nødvendigt inden sproget designes at overveje hvad det vil sige. Det er klart at det er individuelt hvad der gør et sprog let at bruge – det afhænger af hvor god man i forvejen er til at programmere, og hvor mange forskellige sprog man kender. Der eksisterede allerede et „udkast“ til Pesto, nemlig Volanschis oprindelige specialiseringsklasser. Der var ingen krav om at de to sprog skulle være ens, men at et udkast allerede eksisterede, og vel at mærke et udkast som nogen havde gjort sig tanker om [Volanschi98a], gjorde at jeg ikke begyndte helt på bar bund med designet. I starten af processen gjorde jeg meget ud af at overveje den syntaks som Volanschi foreslår, og jeg synes den er god og intuitiv for javaprogrammører, men jeg har ikke ukritisk adopteret Volanschis design. Jeg har lavet små ændringer og selvfølgelig en del tilføjelser, blandt andet efter at have skrevet

nogle specialiseringsklasser og talt med folk med erfaring i partiel evaluering og hørt deres mening, både om syntaks og hvilke konstruktioner der er brug for. Et problem der er let at løbe ind i når man designer et sprog, er at man er så meget inde i sproget at det kan være svært at lade være med at tilføje en masse ny smart syntaks som *lige* giver lidt ekstra funktionalitet – det er vigtigt at finde en balance. Det har jeg været meget påpasselig med under design og udvikling af sproget. Det er klart at hvis der mangler funktionalitet, skal den tilføjes i sproget, men jeg ønsker at programmøren skal have mulighed for at skulle tage stilling til så lidt som muligt. Der findes dog eksempler på ting der kunne være praktisk at kunne udtrykke i Pesto, men som i Pestos nuværende form ikke er muligt. Disse ting vil blive behandlet i kapitel 6.2, og der vil jeg også diskutere hvor stor relevans disse eventuelle tilføjelser har for brugen af Pesto.

Ud over Volanschis oprindelige specialiseringsklasser er et Pesto-lignende sprog blevet brugt i artikler til at præsentere partiel evaluering [Lawall99, Schultz99, Schultz00, Schultz02] hvor forskellige sprogkonstruktioner er foreslået. Der er dog aldrig implementeret en oversætter til disse sprog. Disse konstruktioner er også indgået i mine overvejelserne om synktaksen.

Et andet vigtigt krav der udspringer af at sproget skal være let at bruge, er at det er præcist og entydigt. Der skal ikke være tvivl om hvad en konstruktion i sproget betyder, også selv om man ikke er helt hjemme i Pesto. Det er klart at dette krav også er afhængig af beskuerens referencerammer. For at opfylde dette krav har jeg brugt min generelle viden om og erfaring med og forståelse af sprog, og overvejet hvad der kan misforstås og hvordan, først og fremmest ud fra mine egne referencerammer.

Et sidste, men ikke ubetydeligt krav til Pesto er at det skal være udtryksfuldt. Det skal mindst være muligt at udtrykke de ting som der har været brug for i de artikler der har brugt specialiseringsklasser [Schultz99, Lawall99, Schultz00, Schultz02]. Dog er det vigtigt at dette krav ikke tager over og netop går ud over enkeltheden i sproget. Præcis hvilke valg og overvejelser der er gjort ved de forskellige konstruktioner i sproget, vil blive gennemgået i de følgende kapitler.

4.2 Generel beskrivelse af Pesto

En specialiseringsklasse er en blok der forholder sig til en javaklasse, specialiseringsklassens *rodklasse*. Specialiseringsklassen indeholder et antal prædikater som alle forholder sig til instansvariabler i rodklassen. En specialiseringsklassefil forholder sig til et javaprogram og består af en eller flere specialiseringsklasser hvor én af specialiseringsklasserne skal indeholde specialiseringens *indgangspunkt*. For at give mening skal alle specialiseringsklasserne være internt afhængige. Hvis ikke, gives en advarsel når oversætteren køres, og de klasser der ikke nås har ingen betydning for specialiseringen.

Hver specialiseringsklasse starter med nøgleordet **specclass** (kort for *specialization class*) efterfulgt af navnet på specialiseringsklassen, endnu et nøgleord og navnet på rodklassen som set i tidligere eksempler. Hele grammatikken i Backus-Naur-form kan ses i bilag B. Selve specialiseringsklassen er en *kvasi-invariant* for rodklassen. Som beskrevet i kapitel 3.2 bliver den specialiserede kode udført når kvasi-invarianten er sand, og kvasi-invarianten er sand når alle prædikaterne i specialiseringsklassen er opfyldt. Men før det specialiserede program kan udføres, skal den specialiserede kode genereres og kaldes. Koden genereres af JSpec ud fra oplysningerne i specialiseringsklasserne og Pesto sørger for at kalde koden på det rigtige tidspunkt. Hvordan koden genereres, bliver beskrevet i kapitel 5. Først vil jeg beskrive hvordan den enkelte specialiseringsklasse er opbygget.

4.3 Variabelprædikater

Et variabelprædikat er et prædikat på en instansvariabel i rodklassen og er grundpillen i Pesto. Der eksisterer grundlæggende to former for variabelerklæringer som vil blive beskrevet nedenfor. Fælles for de to er at den instansvariabel de udtrykker noget om skal være defineret i rodklassen eller i en superklasse til rodklassen, og de prædikater der lægges på dem skal være konsistente med instansvariablens type.

Et prædikat i Pesto har faktisk to funktioner. For det første er det en slags tilordning af instansvariablen som den specialiserede kode bruger.[1] For eksempel tilordnes eksponenten *exp* i specialiseringsklassen `Cube` værdien 3, og i den specialiserede kode har instansvariablen *exp* værdien 3. For det andet er prædikatet en slags betingelse for at den specialiserede kode køres i stedet for den generiske – for programmet `Power` betyder det at når `raise` kaldes, kontrolleres det om prædikaterne er opfyldt, i dette tilfælde at *exp* er 3, *inden* metodens krop udføres. Hvis prædikatet er opfyldt køres den specialiserede kode hvor *exp* netop er 3.

Et af kravene til design af sproget var som nævnt at det skulle ligne Java. Men som læseren måske allerede har bemærket, ligner prædikaterne i Pesto ikke javaerklæringer. I Java tilordnes instansvariabler værdier med erklæringen: „*v* = *værdi*“, og en instansvariabels type erklæres som: „*T v*“. Et Pesto-prædikat er også en slags tilordning som beskrevet i forrige afsnit, men denne tilordning benyttes kun hvis prædikatet er sandt, og for at gøre denne forskel tydelig er en anden notation valgt. Selvom prædikaterne ikke nødvendigvis ligner javaudtryk, er sproget „javalignende“ – det er bygget op af klasser, og man kan godt tænke på specialisering som en slags nedarvning hvor specialiseringsklasserne nedarver fra deres rodklasser.

[1] I denne sammenhæng er prædikat ikke et godt ord da et prædikat har en sandhedsværdi, og det kan en tilordning næppe siges at have. Jeg bruger ordet prædikat for at undgå for mange begreber.

Værdierklæring

En *værdierklæring* er et prædikat på en instansvariabels værdi. Værdierklæringer kan erklæres på primitive typer som for eksempel **int**, **char** og **boolean**. Desuden kan erklæringerne også være på String.[2]

Syntaksen for værdierklæringer er:

$$v \texttt{ == } \textit{værdi}\texttt{;} \tag{12}$$

Syntaksen er den samme som for en sammenligning i Java. Et prædikat i Pesto har både rollen som en tilordning og en sammenligning, men det er sammenligningen der er det relevante for brugeren af Pesto, og derfor er det den notation der er valgt. Programmøren skal tænke på sit prædikat som en *vogter* der bliver sat ind i hans kode. Vogteren spørger:

```
if (v == værdi) {
    [specialiseret kode]
} else {
    [oprindelig kode]
}
```
(13)

Afhængig af hvilken værdi v har under udførelsen, vil vogteren vælge hvilken kode der skal udføres. Naturligvis skal vogteren kontrollere alle specialiseringsklassens prædikater. Vogteren gælder ikke under hele programmet, men kun ved indgangspunktet. Vogterens rolle vil blive uddybet i kapitel 5.4.

Som beskrevet i kapitel 3.3, kan de specielle bindingstidstegn bruges til værdierklæringer. At skrive at en variabel er dynamisk (?), er det samme som slet ikke at have prædikatet hvorimod implicit statiske prædikater (!) betyder at værdien vil blive oplyst senere, og først når den faktiske værdi kendes, genereres vogteren.

Typeerklæring

En typeerklæring er et prædikat der siger noget om typen på en instansvariabel: hvis variablen v har typen T, er prædikatet opfyldt. Typeerklæringer kan kun erklæres på instansvariabler der ikke har en primitiv type.

Syntaksen for typeerklæringer er følgende:

$$v\texttt{: } T\texttt{;} \tag{14}$$

Denne konstruktion kendes slet ikke fra Java. I artiklen om Specialization Patterns [Schultz00] bruges javasyntaksen „$T\ v$;“ hvilket i Java betyder at v er af typen T *eller*

[2] Selv om String er en objekttype, opfører den sig i mange sammenhæng som en primitiv type, for eksempel ved tilordninger.

en subtype af T. Jeg har valgt kolon-notationen for at undgå denne tvetydighed og desuden med de samme overvejelser som for værdierklæringer. I Pesto er det interessant at kunne udtrykke at v har *præcis* typen T da det er det der ønskes specialiseret for. I kapitel 6.2 om perspektiverne for Pesto beskrives muligheden for at specialisere for at en instansvariabel har en givet type eller en subtype *eller* dens subtype.

I kapitel 3.3 nævnes bindingstidstegnene ! og ?. Udråbstegnet er ikke den eneste måde at udtrykke at man ved at noget er implicit statisk, og at den faktiske værdi vil blive givet senere. Ved typeerklæringer er erklæringen lidt mere konkret – for værdierklæringer specialiseres der for at en instansvariabel kan have en hvilken som helst værdi. Ved ikke-primitive instansvariabler er det nødvendigt at skrive at instansvariablen kan have én af flere typer $T_1,\ldots,T_n$.

Syntaksen er:

$$v\texttt{:}\ T_1\ \texttt{|}\ T_2\ \texttt{|}\ \ldots\ \texttt{|}\ T_n\texttt{;} \tag{15}$$

hvor | læses som „eller“.

Det er klart ifølge semantikken for Pesto at $T_1,\ldots,T_n$ skal være subtyper af typen af v. Også her siges prædikaterne at være implicit statiske og det skal senere angives præcis hvilken type variablen skal have i specialiseringen. Hvordan det angives, beskrives i kapitel 4.6.

På samme måde som for værdivariabler genererer Pesto en vogter for typeerklæringer. Vogteren kontrollerer at den givne instansvariabel har den rette type.

```
if (v.getClass( ) == T.class) {
   [specialiseret kode]
} else {                                                    (16)
   [generisk kode]
}
```

4.4 Indgangspunktet

Som beskrevet i kapitel 3.2, er det ofte kun et udsnit af et program der ønskes specialiseret. Grunden er at man ønsker så stærk en specialisering som muligt og derfor gerne vil kunne udtale sig om variablers værdi på et bestemt sted i programmet. Hvis en variabel ikke giver mening for et bestemt programpunkt, men gør for et andet, vil det første punkt måske ikke blive specialiseret da hele kvasi-invarianten ikke er opfyldt. Desuden er specialisering af et helt program en dårlig ide da det vil tage alt for lang tid, og derved går noget af ideen ved at specialisere af.

Indgangspunktet angives ved en metodesignatur, det vil sige metodens *modifiers*, dens returtype, navn og eventuelle parametre og deres typer. Metoden kaldes specialiserin-

gens indgangspunkt. Alle metodekald fra denne metode specialiseres. Som nævnt kan der kun angives et indgangspunkt i Pesto.

Det er i Pesto muligt at erklære prædikater på indgangspunktets parametre. Prædikaterne er af samme type som beskrevet ovenfor; enten værdierklæringer eller typeerklæringer. Før hver prædikat skrives nøgleordet **where**. Følgende kode er en modificeret udgave af SpecPower hvor det i stedet er basen der specialiseres med hensyn til:

```
specclass KnownBase specializes Power {
  public int raise(int base) {
    where base == 2;
  }
}
```
(17)

4.5 Arrays

Som et specialtilfælde af typeerklæringer har man i Pesto muligheden for at erklære prædikater på arrays. Prædikaterne bruges på samme måde som typeerklæringer. Man kan specialisere for at en instansvariabel har en bestemt type som i dette tilfælde er en arraytype. Det der er specielt er at det også er muligt at specialisere for længden af et array og for indholdet af arrayet. I specialiseringen af roboteksemplet blev specialiseringen af et array brugt til at specialisere for antallet af observatører og for hvilke observatører subjektet indeholdt:

```
specclass SpecThermo specializes Thermometer {
  observers: Observer[2] = {ForwardController, DirectionController};
}
```
(18)

Her er *observers* et array af Observer af længden to, der som første element indeholder et objekt af typen ForwardController og som andet element indeholder et objekt af typen DirectionController. Begge er, og skal være, subklasser af Observer.

I ovenstående eksempel var mange oplysninger om de faktiske værdier i arrayet kendt, nemlig præcis hvilke typer de enkelte indgange i arrayet havde, men det er ikke altid tilfældet – eller ønskværdigt. Derfor er det selvfølgelig også muligt at udtrykke blot nogle af aspekterne ved et array der skal specialiseres. For det første blot typen:

```
observers: Observer[];
```
(19)

Da kolon i Pesto betyder at variablen har netop denne type og ikke en subtype, betyder kodeeksemplet at *observers* har netop typen Observer[] og ikke en subtype. Mere interessant er det ofte at specialisere for længden på arrayet og dermed folde en eventuel løkke ud. Længden kan erklæres eksplicit statisk, som i kodeeksempel 18 hvor den er

to, eller blot implicit statisk:

```
observers: Observer[!];                                                    (20)
```

Desuden kan indholdet i arrayet specificeres som implicit statisk i stedet for eksplicit statisk[3] som i kodeeksempel 18. Det kan gøres ved at bruge „eller"-konstruktionen fra typeerklæringer:

```
observers: Observer[2] = {ForwardController | DirectionController,
                          ForwardController | DirectionController};        (21)
```

Her vides hvilke observatører der er, men ikke i hvilken rækkefølge de findes i arrayet. De krøllede parenteser kan byttes ud med kantede parenteser, i så fald erklærer man at det samme gælder for alle elementer i arrayet:

```
observers: Observer[!] = [ForwardController];                              (22)
```

Koden specialiseres nu for at alle elementer i *observers* har typen ForwardController. Dette stykke kode ville ikke udløse udførsel af specialiseret kode i eksemplet fra kapitel 3.4 da det ikke gælder at alle elementer i arrayet *observers* er ForwardControllers. Tilsvarende kan man bruge „eller"-konstruktionen i indholdet af et array:

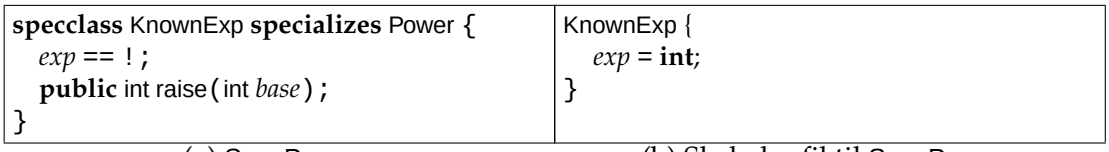
```
observers: Observer[!] = [ForwardController | DirectionController];        (23)
```

Her analyseres der for at alle elementerne har samme type, enten ForwardController eller DirectionController, modsat kodeeksempel 21 hvor elementerne kunne have forskellige typer. I kodeeksempel 22 er længden i arrayet angivet som implicit statisk, og det skal dermed først tilkendegives senere hvor langt arrayet er. I kodeeksempel 21 giver det ikke mening at have en dynamisk eller implicit statisk længde da man ved at angive det eksplicitte indhold i arrayet allerede har angivet længden – antallet af elementer i de krøllede parenteser skal være det samme som den angivne længde på arrayet – hvis ikke gives en fejl. Hvis længden alligevel kun angives som dynamisk eller implicit statisk, giver det ikke en fejl i oversætteren, men den faktiske længde bruges som den er givet ved antallet af elementer i indholdet.

4.6 Erklæring af de eksakte værdier

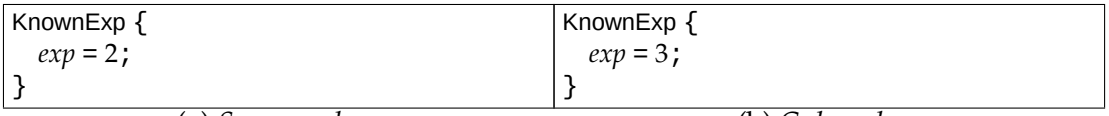
Som det fremgår af tidligere eksempler, medfører brugen af implicit statiske prædikater behovet for at kunne erklære de eksakte specialiseringsværdier efter analysen. Erklæringen af de eksakte værdier sker i en værdifil der efterfølgende læses under specialiseringen som gør brug af de eksakte værdier. For at hjælpe brugeren til at erklære

[3] Her bruges eksplicit statisk som eksplicit ud fra hvad der i dette prædikat kan vides om variablen, det vil sige længden. En arrayvariabel kan dermed godt betegnes som eksplicit statisk selv om den indholder dynamiske variabler.



(a) SpecPower (b) Skabelonfil til SpecPower

Figur 4.1: Specialiseringsklasse samt skabelonfil til den simple eksponentialfunktion.



(a) *Square.values* (b) *Cube.values*

Figur 4.2: Modificerede skabeloner fra SpecPower

værdierne, skriver Pesto hvilke værdier det er nødvendigt at erklære og hvilken type de eksakte værdier skal have i en skabelonfil. Skabelon- og værdifilen er skrevet i et format der gør det let at overskue hvor de eksakte værdier skal indsættes og som desuden er let at parse når værdierne skal bruges i specialiseringsfasen. Formatet ligner specialiseringsklasserne med specialiseringsklassens navn efterfulgt af en blok hvori specialiseringsklassens variabler er skrevet. De steder hvor der ønskes nye værdier står typen på den værdi der ønskes. Figur 4.1 (a) viser specialiseringsklassen fra figur 3.2 til Power-eksemplet fra kapitel 3, og 4.1 (b) den skabelon som Pesto udlæser med KnownExp som input.

En af fordelene ved at bruge bindingstidstegn og implicit statiske variabler i stedet for eksplicit statiske er at det er muligt at vente med at angive de endelige værdier som måske ikke kendes af den programmør der skriver specialiseringsklasserne. Men den største fordel er at det er muligt at have flere forskellige specialiseringsværdier til én generel bindingstidsanalyse. Dette er en fordel da det er analysen der er den tidskrævende del af den partielle evaluering. I figur 4.2 ses to forskellige værdierklæringer til Power-eksemplet hvor filen fra figur 4.1 (b) er modificeret. De to giver specialiseret kode til henholdsvis beregningen af arealet af et kvadrat og rumfanget af en terning. Afhængig af værdien af *exp* sørger Pesto for at den rigtige specialiserede kode udføres ved at bruge vogtere. Brugeren skal blot når han kalder specialiseringen, gøre Pesto opmærksom på hvilke versioner der ønskes specialiseret for. Jo flere forskellige specialiseringer af koden, jo flere værdier vil udløse den specialiserede kode, men jo længere tager specialiseringen, og jo flere betingelser skal kontrolleres for at finde ud af hvilken kode der skal udføres. Koden kommer derved også til at fylde mere, da den både skal indeholde den generiske kode og den specialiserede kode samt vogterne.

4.7 Referencer mellem specialiseringsklasser

I roboteksemplet fra kapitel 3.4 er der i specialiseringsklasserne typeerklæringer på formen:

$$v\text{: } S; \tag{24}$$

hvor S er navnet på en specialiseringsklasse (se figur 3.7). Det vil sige at man fra en specialiseringsklasse kan *referere* til en anden specialiseringsklasse der ligger i samme fil. Alle steder hvor man i Pesto kan skrive en klasse, kan man skrive en specialiseringsklasse der selvfølgelig skal være konsistent med den oprindelige type for den instansvariabel man ønsker at specialisere.

Ved at bruge referencer mellem specialiseringsklasser kan man foretage en stærkere specialisering som i roboteksemplet. Specielt i eksempler med finkornet objektstruktur hvor funktionaliteten er delt ud på flere objekter, er referencer et godt værktøj.

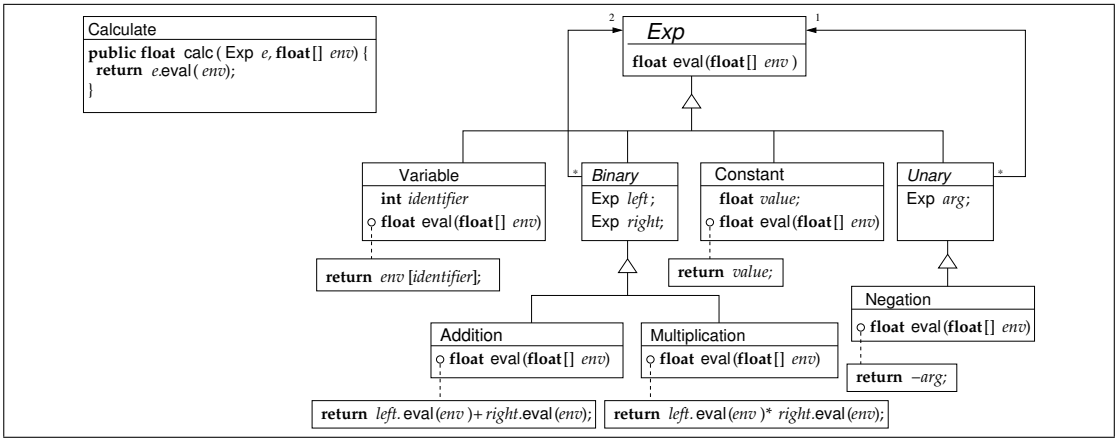
Referencerne programmeres som typeerklæringer hvor klassenavnet er byttet ud med navnet på en specialiseringsklasse defineret i samme fil som den specialiseringsklasse hvori referencen er.

Referencer mellem specialiseringsklasser bruges hvis man kender lidt mere end blot typen på den klasse en instansvariablen refererer til. Som skrevet tidligere er der flere eksempler på artikler der bruger specialiseringsklasser til at beskrive automatisk programspecialisering, dog uden at have en konkret implementation, kun en løs grammatik. I artiklerne har det vist sig utrolig nyttigt at kunne referere mellem specialiseringsklasser [Schultz99, Schultz00, Schultz02].

4.8 Rekursive erklæringer

En naturlig følge af referencer mellem specialiseringsklasser er referencer til specialiseringsklassen selv.

Et program der benytter referencer til sin egen klasse kan for eksempel være et program der beregner aritmetiske udtryk. Klassediagrammet for et sådant program ses i figur 4.3. Et udtryk (*Exp*) kan enten være et binært udtryk (*Binary*) eller et udtryk med ét deludtryk (*Unary*). *Exp*, *Binary* og *Unary* er abstrakte klasser. Et binært udtryk består af et venstre og et højre deludtryk som kan være et hvilket som helst udtryk. Et udtryk kan desuden være en konstant (Constant) eller en variabel (Variable). Af binære udtryk bruger jeg multiplikation (Multiplication) og addition (Addition), mens negation (Negation) er et udtryk med ét deludtryk. Klassen Exp har en metode, eval, som beregner resultatet af udtrykket givet et array af værdier til mulige variabler. Man kalder beregningen gennem klassen Calculate via metoden calc der givet et udtryk og et array af variabel-



Figur 4.3: Klassediagram for beregning af aritmetiske udtryk.

værdier returnerer værdien af udtrykket.

Et eksempel på en specialisering af programmet ses i figur 4.4. Programmets main-metode ligger i *Expression.java*, og det er her de forskellige objekter initialiseres. Metoden calc er specialiseringens indgangspunkt – det er herfra beregningen af udtrykket starter. For at kunne bruge analysen til så mange forskellige udtryk som muligt, er denne specialisering generel, og variablerne er implicit statiske. Af tekniske grunde er det ikke muligt i Pesto at specialisere for at noget har en abstrakt type så det udtryk calc kaldes med, skal enten være en multiplikation, en addition, en negation, en konstant eller en variabel. Alle disse klasser har specialiseringsklasser, og det er dem der refereres til i AnyExpression. Hvert af de to binære udtryk er specialiseret med hensyn til at henholdvis deres højre og venstre udtryk kan være hvilket som helst af de konkrete klasser. Det ses at både SpecMult og SpecAdd refererer til sig selv – en multiplikation kan selvfølgelig godt bestå af en multiplikation i højre udtryk og en til i venstre, så med denne skabelon kan jeg konstruere et hvilket som helst udtryk bestående af multiplikationer og additioner, og det samme gælder for SpecNeg.

Som et eksempel på hvordan en specialiseringskontekst skrevet før Pesto var implementeres ses i figur 4.5 en JSpec-kontekst skrevet i hånden. Konteksten svarer nogenlunde til AnyExpression borset fra at den er til et program som beregner udtryk på heltal i stedet for floats. Både at skrive og læse denne kontekst er nærmest umuligt med mindre man er meget godt inde i både Java og JSpec. I konteksten kan brugeren også godt angive de faktiske værdier efter analysen, men som endnu et javaprogram.

Figur 4.6 viser den skabelon Pesto udskriver og som skal ændres til at indholde de værdier programmet ønskes specialiseret for. De to variabler *e* og *env* er argumenter

```
specclass AnyExpression specializes Calculate {
  public float calc(Exp e, float[] env) {
    where e : SpecMult|SpecAdd|SpecConst|SpecVar|SpecNeg;
    where env : float[2];
  }
}
specclass SpecMult specializes Multiplication {
  left : SpecMult|SpecAdd|SpecConst|SpecVar|SpecNeg;
  right : SpecMult|SpecAdd|SpecConst|SpecVar|SpecNeg;
}
specclass SpecAdd specializes Addition {
  left : SpecMult|SpecAdd|SpecConst|SpecVar|SpecNeg;
  right : SpecMult|SpecAdd|SpecConst|SpecVar|SpecNeg;
}
specclass SpecNeg specializes Negation {
  arg : SpecMult|SpecAdd|SpecConst|SpecVar|SpecNeg;
}
specclass SpecConst specializes Constant {
  value == !;
}
specclass SpecVar specializes Variable {
  identifier == !;
}
main file : Expression.java
entrypoint overwrite : (S,D)
```

Figur 4.4: Specialiseringsklasser til en generel specialisering af aritmetiske udtryk.

```
import fr.irisa.compose.jspec.StaticValue;
import fr.irisa.compose.jspec.DynamicValue;
public class _JSpec_Context {
  public static Exp e;
  public static int[] x;

  // Wrapper method to call main
  public static void main(String[] argv) {
    Main.main(argv);
  }

  // Method for setting static fields
  public static void set(Exp _e, int[] _x) {
    e = _e;
    x = _x;
  }

  // Method for setting analysis context
  public static void setAnalysisContext() {
    int[] i = DynamicValue.get_boolean()?new int[DynamicValue.get_int()]:new
int[DynamicValue.get_int()];
    i[0] = DynamicValue.get_int();
    set( _JSpec_Context.buildArbitraryExp(), i );
  }

  public static Exp buildArbitraryExp() {
    Exp e = null;
    while( StaticValue.get_boolean() ) {
      int x = StaticValue.get_int();
      if( x==0 ) e = new Add( e, e );
      else if( x==1 ) e = new Constant( StaticValue.get_int() );
      else if( x==2 ) e = new Mul( e, e );
      else if( x==3 ) e = new Neg( e );
      else e = new Var( StaticValue.get_int() );
    }
    return e;
  }

  public static void setSpecializationContext() {
    // User must supply body, -sctx, default is setAnalysisContext() like here
    set(Main.buildExp(), (int[])DynamicValue.get_object());
  }
}
```

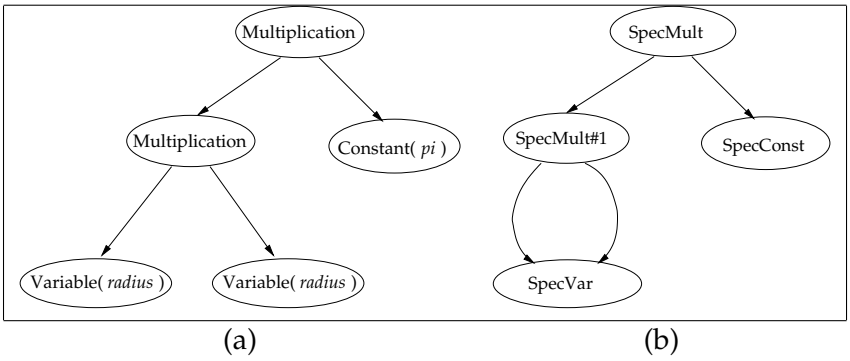
Figur 4.5: Et eksempel på en kontekst skrevet i hånden uden Pesto.

```
AnyExpression {
    calc(e) : SpecMult | SpecAdd | SpecConst | SpecVar | SpecNeg;
    calc(env) : float[int];
}

SpecMult {
    left : SpecMult | SpecAdd | SpecConst | SpecVar | SpecNeg;
    right : SpecMult | SpecAdd | SpecConst | SpecVar | SpecNeg;
}

SpecAdd {
    left : SpecMult | SpecAdd | SpecConst | SpecVar | SpecNeg;
    right : SpecMult | SpecAdd | SpecConst | SpecVar | SpecNeg;
}

SpecNeg {
    arg : SpecMult | SpecAdd | SpecConst | SpecVar | SpecNeg;
}

SpecConst {
    value = float;
}

SpecVar {
    identifier = int;
}
```

Figur 4.6: Skabelonfil til eksakte specialiseringsværdier.

```
AnyExpression {
    calc(e) : SpecMult;
    calc(env) : float[1];
}

SpecMult {
    left : SpecMult#1;
    right : SpecConst;
}

SpecMult#1 {
    left : SpecVar;
    right : SpecVar;
}

SpecConst {
    value = 3.1416F;
}

SpecVar {
    identifier = 0;
}
```

Figur 4.7: *Area.values* – de faktiske værdier til beregning af arealet af en cirkel.



Figur 4.8: Strukturen af et program der beregner arealet af en cirkel.

til metoden calc hvilket angives ved metodenavnet og parameternavnet direkte efter i parentes. Dette er for at undgå tvetydigheder ved eventuelle navnesammenfald.

Det er som nævnt tidligere ikke meningen at filen skal læses direkte af brugeren, men derimod vises gennem et værktøj hvor det ikke er muligt at ændre noget i filen der ikke må ændres. Derfor er det ikke så afgørende at formatet ikke er helt entydigt – for eksempel linjen: `calc(env) : float[int];` hvor man skal ændre int, men ikke float da float er typen på arrayet og int er den implicit statiske længde.

I figur 4.7 vises en modificeret udgave af skabelonen som beskriver en specialisering af en beregning der skal foretages af Calculate for at beregne arealet af en cirkel. Det udtryk der specialiseres for beregner arealet af en cirkel og strukturen på programmet ses i figur 4.8 (a). I udtrykket er der to (forskellige) multiplikationer, det vil sige to objekter

der har typen Multiplication og har forskellige deludtryk. Som det ses på figur 4.8 (a), har den ene multiplikation en anden multiplikation som venstre deludtryk og som højre en konstant. Den anden multiplikation har en variabel i både højre og venstre deludtryk. Disse Variabel-objekter indholder den samme værdi, nemlig cirklens radius. Der skal altså være to forskellige specialiseringsklasser til Multiplication hvilket løses ved at nummerere dem. Der er også to Variabel-objekter i udtrykket, men for dem gælder det at det er det samme der ønskes specialiseret for og derfor refereres blot til den samme specialiseringsklasse. Som det ses i figur 4.7, bruges klaf (#) og et nummer som intern notation til at skelne de forskellige specialiseringsklasser. Figur 4.8 (b) viser strukturen af specialiseringsklasserne.

Når *Area.values* er specificeret, kan specialiseringen af programmet foretages. Den specialiserede kode vil herefter blive kaldt hvis calc kaldes med et udtryk med strukturen vist i figur 4.8 (a). Hvis arealet af en cirkel beregnes med et udtryk som ikke har samme struktur, for eksempel hvis $radius^2$ beregnes i den første multiplikations højre deludtryk, vil vogteren ikke udløse kald til den specialiserede kode.

Referencer til de konkrete objekter

Jeg skrev i ovenstående at der i specialiseringskonteksten specialiseres for to forskellige multiplikationer og én variabel som bliver peget på af forskellige instansvariabler (se figur 4.8 (b)). Rent praktisk sker der det i den vogter Pesto genererer at den kontrollerer om de relevante instansvariabler har de korrekte værdier. Så når jeg siger at der kun er et specialiseret variabelobjekt, betyder det blot at de objekter der refereres til indeholder en instansvariabel med samme værdier. Det er ikke nødvendigvis det samme objekt, men kan godt være det. Resultatet af specialiseringen vil være det samme uanset om der i programmet er to Variabel-objekter med den samme værdi (*identifier* = 0) eller om det er det samme objekt. Det er bestemt en interessant problemstilling at kunne specialisere for at to referencer peger på samme objekt, eller netop ikke gør hvilket jeg vil berøre i kapitel 6.2 om perspektiverne for deklarativ specialisering.

4.9 Sammenfatning

Jeg begyndte dette kapitel med at gøre rede for hvilke overvejelser jeg har gjort mig under design og udviklingen af Pesto. At konstruere et sprog er således en iterativ proces. Jeg startede med en grammatik og byggede ud fra den en oversætter. Derefter brugte jeg sproget, og derved dukkede der interessante problemområder op som, afhængig af relevansen og brugbarheden, blev tilføjet til sproget, og oversætteren blev udbygget og så videre. I denne proces er jeg kommet frem til de sprogkonstruktioner jeg i dette kapitel har gjort beskrevet, og jeg har desuden givet eksempler på hvordan sproget bruges.

I kapitel 5 vil jeg redegøre for hvordan Pesto er opbygget som system og hvordan den interne sammenhæng mellem Pesto og det program der foretager specialiseringen er.

5 Design af systemet

Jeg har i de forgående kapitler beskrevet sproget Pesto og fortalt hvordan Pesto er en *front end* til JSpec. I dette kapitel vil jeg beskrive opbygningen af Pesto og grænsefladen til JSpec.

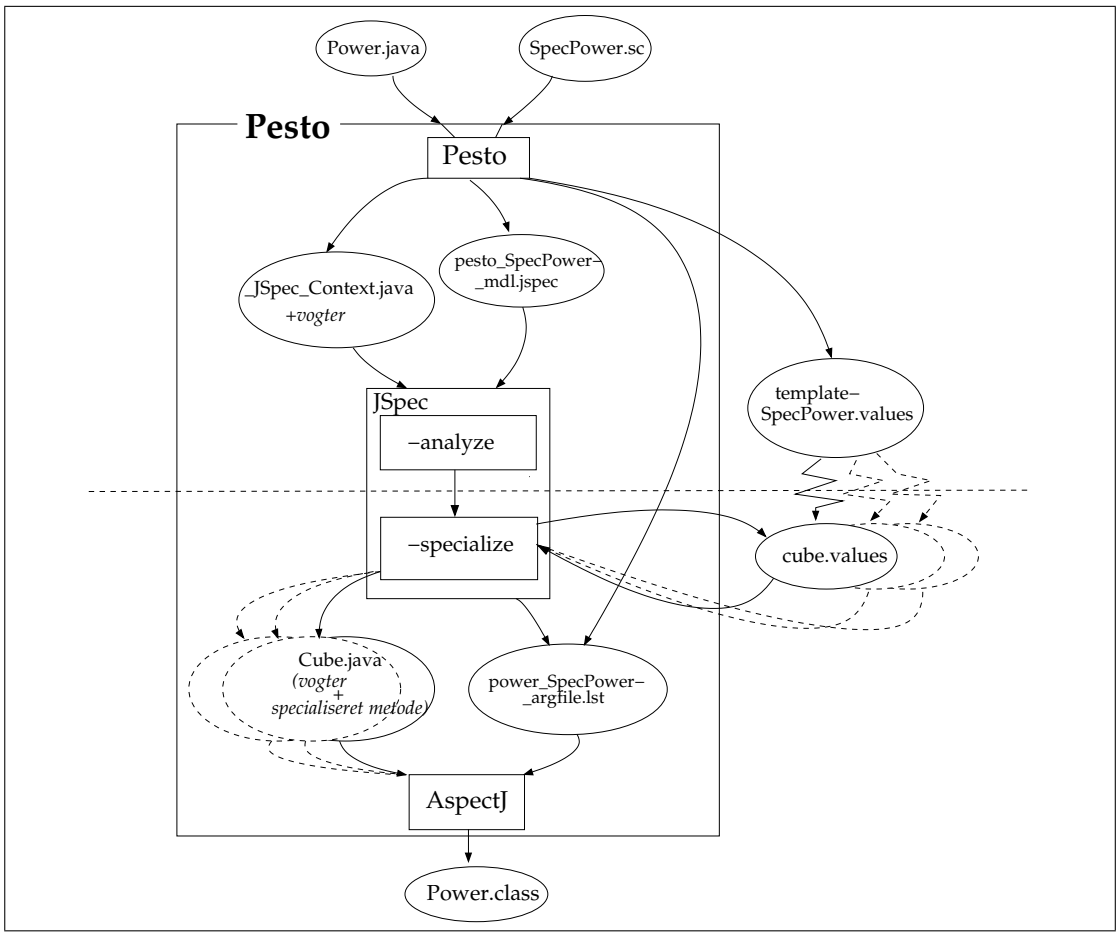
Figur 5.1 viser strukturen af Pesto og sammenhængen med JSpec. Figuren er en mere detaljeret udgave af figur 3.1, og som eksempelprogram i figuren har jeg valgt `Power` og en specialiseringsklasse `SpecPower`. De punkterede ovaler repræsenterer eventuelle flere forskellige specialiseringsværdier.

For at lette arbejdet med at specificere oversættelsen fra Pesto til JSpec har jeg skrevet en semantik over oversættelsen. Hele semantikken kan ses i bilag C. Semantikken i bilaget følger ikke helt den konkrete semantik for JSpec da semantikken i bilaget deler de to kontekster, analysekonteksten og specialiseringskonteksten, op i to forskellige klasser i stedet for to metoder i én klasse. Bortset fra denne afvigelse er semantikken korrekt, om end en smule forkortet – for overskuelighedens skyld har jeg ikke medtaget alle aspekter af Pesto og for eksempel valgt blot at bruge `int` som repræsentant for alle primitive typer. I semantikken i bilaget er det for eksempel kun muligt at have indgangspunktmetoder med `int` som returtype på trods af at Pesto understøtter alle javatyper, men det ville blive for omfattende at beskrive den fulde semantik og ikke give bedre forståelse. I de følgende kapitler vil jeg bruge dele af semantikken som er relevant for kapitlet.

5.1 Interface til JSpec

JSpec er en automatisk programspecialisator til javaprogrammer (en detaljeret beskrivelse af JSpec kan ses i kapitel 2.3) som genererer den specialiserede kode ud fra oplysninger om programkonteksten som brugeren giver til JSpec. JSpec sørger ikke for at koden bliver kaldt på det rigtige sted, men genererer den specialiserede metode som et aspekt der indsættes i rodklassens klassefil ved hjælp af AspectJ.

JSpec bruges traditionelt ved at skrive en specialiseringskontekst i Java. Specialiseringskonteksten er en javaklasse der skal hedde `_JSpec_Context`. Klassen skal mindst have de to metoder `setAnalysisContext` og `setSpecializationContext`. De to metoder kaldes ved



Figur 5.1: Pestos samlede struktur.

henholdsvis analysen og specialiseringen af programmet. Ud over de to metoder er der et antal instansvariabler der er obligatoriske i _JSpec_Context; *_this* er en instans af den rodklasse hvor specialiseringen udspringer, det vil sige rodklassen til den specialiseringsklasse hvor indgangspunktet befinder sig. Når specialiseringen er færdig, skal *_this* indeholde alle de specialiserede værdier. Desuden erklæres parametrene i indgangspunktet som instansvariabler i _JSpec_Context for at kunne tilordne værdier til dem statisk. Ydermere skal hver specialiseringsklasse have sin egen instans både i analysekonteksten og specialiseringskonteksten, disse skal hedde *inst_<specialiseringsklassenavn>*. Klassen _JSpec_Context skal indeholde en main-metode der kalder main-metoden i det program der skal specialiseres. JSpec skal desuden have en konfigurationsfil der indeholder parametre til JSpec. Det er den fil der på tegningen hedder *pesto_SpecPower_mdl.jspec* og som genereres af Pesto hvilket vil blive beskrevet i kapitel 5.5. Bilag D viser en fuld kontekst til JSpec for eksemplet der beregner aritmetiske

```
specclass BinCube specializes BinPower {
   exp == 3;
   op: Mult;
   public int raise(int base);
}
main file: Main.java
```

Figur 5.2: Specialiseringsklassen BinCube.

udtryk fra kapitel 4.8 genereret af Pesto.

JSpec benytter en javapakke, fr.irisa.compose.jspec, specielt designet til programspecialisering med JSpec og som indholder klasserne DynamicValue og StaticValue som har metoder til analysefasen i specialiseringen.

Kort sagt er både analysedelen og specialiseringsdelen i _JSpec_Context en række tilordninger af værdier som skal bruges til at generere den specialiserede kode. I analysedelen er det afgørende om værdier er statiske eller dynamiske, ikke hvad deres faktiske værdier er. Det er her klasserne StaticValue og DynamicValue bruges. De indeholder metoderne get_T() hvor T er en primitiv type, for eksempel int, samt get_object() som bruges til at erklære dynamiske objekttyper.

Kapitel 5.2, 5.3 og 5.5 giver en beskrivelse af de forskellige dele som Pesto genererer for at kunne bruges af JSpec.

5.2 Analysekontekst

Analysefasen har to formål: bindingstidsanalyse og aliasrelationsanalyse. Aliasrelationer er relationer mellem et objekt og dets mulige typer, det vil sige at et objekt har relationer/referencer til de typer objektet kunne tænkes at have under udførslen. Bindingstidsanalysen afgør om en instansvariabel er dynamisk eller statisk på udførselstidspunktet. I de følgende kapitler vil det fremgå hvad de to analyser dækker.

Jeg har delt gennemgangen af analysekonteksten op i to eksempler; et der håndterer de simple prædikattyper i Pesto og et mere kompliceret med rekursive erklæringer.

Kontekseksempel til en binær eksponentialfunktion

Eksemplet bygger på eksponentialfunktionen, men en udvidet udgave hvor man kan vælge om man vil have *basis* ganget eller adderet med sig selv *eksponent* gange. Programmet hedder BinPower for *binary power*, og tager som argument en eksponent som også Power gjorde det. BinPower har en instansvariable *op* som er den operation funktio-

```
public static BinPower _this;
public static int base;
public static void setAnalysisContext() {
  BinPower inst_BinCube = new BinPower();
  // Predicates in BinCube
  inst_BinCube.exp = StaticValue.get_int();
  inst_BinCube.op = new Mult();
  // Unused variables in BinCube
  ((Mult)(inst_BinCube.op)).neutral = (int)DynamicValue.get_int();
  base = DynamicValue.get_int();
  _this = inst_BinCube;
}
```

Figur 5.3: Analysekonteksten for specialiseringsklassen BinCube.

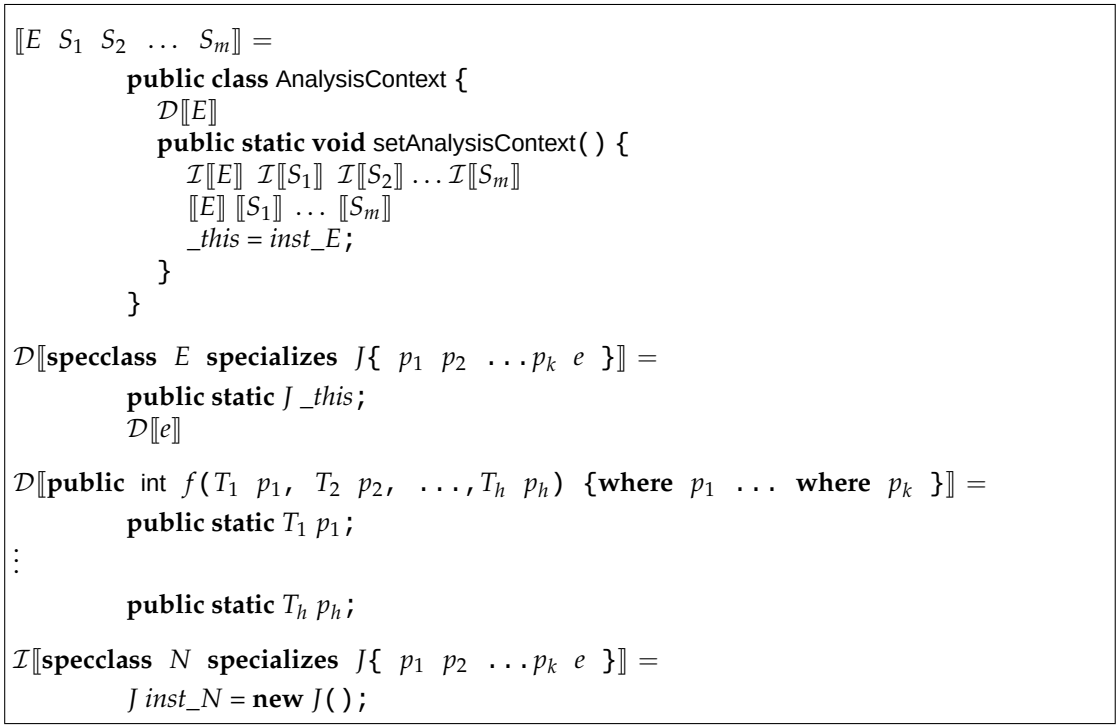
nen ønskes kørt på. Den binære operation har en neutral værdi der for multiplikation er 1, og for addition 0. Hele koden for programmet findes i bilag E. Figur 5.2 ligner den kendte specialiseringsklasse Cube, men her skal der også blot specialiseres for at *op* er et objekt med typen Mult. Figur 5.3 er den tilhørende analysekontekst.

Semantikken for analysekonteksten ses i sin helhed i bilag C.1. T er mængden af typer, både specialiseringsklasser (S) og javaklasser (J).

Det første af semantikken ses i figur 5.4. Den første regel tager en specialiseringsklassefil bestående af en indgangspunktsspecialiseringsklasse (E) og et antal specialiseringsklasser ($S_1, \ldots, S_m$). De oversættes til klassen AnalysisContext[1] som indeholder metoden setAnalysisContext. Den første funktion, $\mathcal{D}$, sørger for erklæringen af de variabler JSpec skal bruge, nemlig *_this* samt alle parametrene fra indgangspunktet. Funktionen $\mathcal{I}$ giver erklæringen af alle instanserne af specialiseringsklasserne.

I analysekonteksten i figur 5.3 ses at *_this* er en instans af BinCubes rodklasse, BinPower. Det er i denne instans de tilordnede værdier ender. Desuden er raises parameter *base* erklæret da parameteren *base* ikke eksisterer direkte i klassen BinPower, og derfor erklæres parametre selvstændigt. JSpec kan efterfølgende bruge parametertilordningerne selv om de ikke er en del af *_this*. Inde i metoden setAnalysisContext erklæres en instans af hver specialiseringsklasse – i BinPower-eksemplet kun den ene. Denne instans indeholder naturligvis alle instansvariabler i rodklassen og da specialiseringsklassen kun indeholder de samme variabler, er det disse instansvariabler der skal tilordnes bindingstider. For at _JSpec_Context kan læses disse instansvariabler skal de være erklæret *public* hvilket er et af de få krav Pesto stiller til programmet. Dermed er omgivelserne

[1] Som beskrevet indledningsvist, eksisterer analysekonteksten ikke som en klasse for sig i den faktiske implementation, men det er pænere at beskrive det således i semantikken.



Figur 5.4: Semantikstump for analysekonteksten. Selve metoden setAnalysisContext erklæres og de instansvariabler JSpec kræver erklæres.

$[\![$**specclass** E **specializes** J`{` p_1 p_2 `...` p_k e `}`$]\!] =$
$\quad [\![p_1]\!](E)\ [\![p_2]\!](E) \ldots [\![p_k]\!](E)\ [\![e]\!]$

$[\![$**specclass** S_i **specializes** J_i `{` p_1 p_2 `...` p_k `}`$]\!] =$
$\quad [\![p_1]\!](S_i)\ [\![p_2]\!](S_i) \ldots [\![p_k]\!](S_i)$

$[\![$**public** int f`(`r_1`,` r_2`,...,` r_h`)` `{`**where** p_1 `...` **where** p_k `}`$]\!] =$
$\quad [\![p_1]\!](\epsilon)\ [\![p_2]\!](\epsilon) \ldots [\![p_k]\!](\epsilon)$

$[\![v$ `==` `?`$]\!](L) =$
$\quad lhs(L, v)$ = DynamicValue.get_int`()`;

$[\![v$ `==` `!`$]\!](L) =$
$\quad lhs(L, v)$ = StaticValue.get_int`()`;

$[\![v$ `==` $number]\!](L) =$
$\quad lhs(L, v)$ = StaticValue.get_int`()`;

$[\![d]\!](L) =$
$\quad lhs(L, d)$ = DynamicValue.get_int`()`;
$\quad$ for enhver primitiv instansvariabel d i Ls rodklassen som ikke er brugt i L

$[\![v$`:`$J]\!](L) =$
$\quad lhs(L, v)$ = **new** J`()`;

$lhs(C, v) = inst\_C.v$
$lhs(\epsilon, v) = v$

Figur 5.5: Semantikstump for analysekonteksten. Tilordning af bindingstiderne for „simple“ prædikater.

sat, og tilordning af aliasrelationer og bindingstider kan begynde.

Semantikken for tilordningen af simple analyseværdier ses i figur 5.5 og i bunden af filen ses en funktion *lhs (left hand side)* der afgør om variablen er en instansvariabel eller en parametervariabel og dermed om venstre side af lighedstegnet er et direkte kald til variablen (parameteren) eller om den skal kaldes gennem dens instansklasse. De første tre regler viser blot at bindingstiderne for hver prædikat tilordnes for sig. Derefter gennemgås de forskellige variabelprædikater et ad gangen. For værdierklæringerne tilordnes enten den statiske eller den dynamiske værdi. Som nævnt er der ikke forskel på implicit statisk og eksplicit statisk i bindingstidsanalysen – begge tilordnes bindingstiden StaticValue.get_T som *exp* i BinPower. „Simple“ typeerklæringer hvor en variabel specialiseres med hensyn til at have en bestemt javatype som *op* i BinCube, håndteres ved at tilordne instansen af variablen et nyt objekt af den ønskede javatype, J, hvilket er

```
specclass AnyExpression specializes Calculate {
  public float calc(Exp e, float[] env) {
    where e: SpecMult|SpecConst|Variable;
    where env: float[!];
  }
}
specclass SpecMult specializes Multiplication {
  left: SpecMult|SpecConst|Variable;
  right: SpecMult|SpecConst|Variable;
}
specclass SpecConst specializes Constant {
  value == !;
}
main file: Expression.java
```

Figur 5.6: Forsimplede specialiseringsklasser til aritmetiske udtryk.

grunden til at alle de klasser der bruges i specialiseringen skal indeholde en default constructor. Hermed er instansvariablens *aliasrelation* givet til at være *J*. Instansvariablens bindingstid er desuden statisk da det er givet hvilken type den har på udførselstidspunktet. Som bekendt kan en typeerklæring også erklæres som implicit statisk hvilket jeg vil beskrive i næste kapitel ved hjælp af et lidt større eksempel. I analysekonteksten til BinCube ses at de instansvariabler og parametre i rodklassen der ikke bliver brugt i specialiseringsklassen også tilordnes bindingtider. I BinCube drejer det sig om parameteren *base*. Når en værdi ikke er statisk, er den dynamisk, det vil sige at alle værdier der ikke er prædikater på er dynamiske, og derfor tilordnes *base* den dynamiske bindingstid. Som det ses i analysekonteksten, er der endnu en ubrugt instansvariabel, *neutral*, fra klassen Mult eller rettere fra Mults superklasse BinOp. Det er prædikatet på *op* der udløser denne dynamiske tilordning da alle instansvariabler i Mult også skal tilordnes den dynamiske værdi for at angive at objektet er dynamisk.

Kontekseksempel til beregning af aritmetiske udtryk

Figur 5.6 er specialiseringsklasser til beregning af aritmetiske udtryk fra kapitel 4.8. Jeg har simplificeret eksemplet en smule for at gøre konteksten mere overskuelig ved at undlade at bruge klasserne Addition og Negation og udelade specialiseringen af klassen Variable. Derved kan jeg konstruere aritmetiske udtryk ved at bruge multiplikation af konstanter og variabler. Figur 5.7 viser analysekonteksten til specialiseringsklasserne, og det ses at konteksten for AnyExpression er mere kompliceret end den for BinPower da AnyExpression indeholder implicit statiske typeerklæringer og, eventuelt selvreferende, referencer mellem specialiseringsklasser. Semantikken for disse konstruktioner

```
public static Calculate _this;
public static Exp e;
public static float[] env;
public static void setAnalysisContext() {
  Calculate inst_AnyExpression = new Calculate();
  Multiplication inst_SpecMult = new Multiplication();
  Constant inst_SpecConst = new Constant();
  // Predicates in AnyExpression
  if (StaticValue.get_boolean()) {
    e = inst_SpecMult;
  } else {
    if (StaticValue.get_boolean()) {
      e = inst_SpecConst;
    } else {
      e = new Variable();
    }
  }
  env = new float[StaticValue.get_int()];
  env[StaticValue.get_int()] = DynamicValue.get_float();
  // Predicates in SpecMult
  if (StaticValue.get_boolean()) {
    inst_SpecMult.left = inst_SpecMult;
  } else {
    if (StaticValue.get_boolean()) {
      inst_SpecMult.left = inst_SpecConst;
    } else {
      inst_SpecMult.left = new Variable();
    }
  }
  if (StaticValue.get_boolean()) {
    inst_SpecMult.right = inst_SpecMult;
  } else {
    if (StaticValue.get_boolean()) {
      inst_SpecMult.right = inst_SpecConst;
    } else {
      inst_SpecMult.right = new Variable();
    }
  }
  // Predicates in SpecConst
  inst_SpecConst.value = StaticValue.get_float();
  // Unused variables in AnyExpression
  ((Variable)(e)).identifier = (int)DynamicValue.get_int();
  // Unused variables in SpecMult
  ((Variable)(inst_SpecMult.left)).identifier = (int)DynamicValue.get_int();
  ((Variable)(inst_SpecMult.right)).identifier = (int)DynamicValue.get_int();
  // Unused variables in SpecConst
  _this = inst_AnyExpression;
}
```

Figur 5.7: Analysekontekst til AnyExpression.

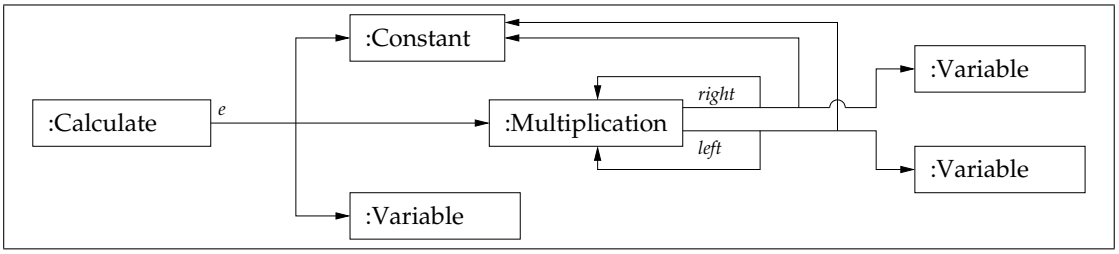
$[\![v : S]\!](L) =$
$\quad lhs(L, v) = inst\_S;$

$[\![v : \ T_1 \mid T_2 \mid \ldots \mid T_k;]\!](L) =$
$\quad$ **if** (StaticValue.get_boolean( )) {
$\quad\quad [\![v : T_1]\!](L)$
$\quad$ } **else** {
$\quad\quad [\![v : T_2 \mid T_3 \mid \ldots \mid T_k]\!](L)$
$\quad$ }

$[\![d]\!](L) =$
$\quad lhs(L, d) =$ DynamicValue.get_object( );

for enhver instansvariabel d af en ikke-primitiv type i Ls rodklassen som ikke er brugt i L

Figur 5.8: Semantikstump for analysekonteksten. Tilordning af bindingstider og aliasrelationer for „avancerede“ prædikater.



Figur 5.9: Aliasrelationer for AnyExpression.

ses i figur 5.8.

I dette eksempel indeholder specialiseringen flere specialiseringsklasser, og for hver af disse erklæres en instansvariabel af samme type som specialiseringsklassens rodklasse. Disse instanser indeholder efter endt analyse de tilordninger der erklæres i specialiseringen.

Den første regel i semantikken i figur 5.8 håndterer „simple“ typeerklæringer til en specialiseringsklasse. I analysekonteksten til AnyExpression ses et eksempel i første gren i den første **if**-sætningen, hvor e tilordnes *inst_SpecMult*, og dermed erklæres det ikke blot at e har typen Multiplication, men også de bindingstider der gælder for variablerne i SpecMult.

Som nævnt tidligere er en instansvariabels aliasrelationer de symbolsk repræsenterede objektinstanser variablen kan have på udførselstidpunktet. Under analysen vil den

værdi der beregnes på hvert programpunkt „**new** *T*()“ i analysekonteksten repræsentere alle de objekter der kan allokeres på programpunktet. Så når jeg i AnyExpression specialiserer for at *e* har en af typerne Multiplication, Constant eller Variable, er disse klasser aliasrelationerne for *e*. I analysekonteksten repræsenteres denne dynamiske information af if-sætningen „**if** (StaticValue.get_boolean())“ som angiver at under udførelsen, vil det være statisk afgjort hvilken af grenene i betingelserne der bliver brugt. Under analysen vil hver gren i betingelsen der skal tilordnes en specialiseringsklasse tilordne en *repræsentation* af en objekttype til instansvariablen. I forhold til eksemplet vil det sige at når *inst_SpecMult.left* tilordnes *inst_SpecConst*, og *inst_SpecMult.right* også tilordnes *inst_SpecConst*, er det ikke det samme objekt de tilordnes, men en repræsentation af objektet, og når de begge tilordnes **new** Variable(), er det forskellige instanser af klassen. Under specialiseringsfasen tilordnes det konkrete objekt, men under analysen har JSpec blot brug for at vide hvilken „slags“ objektet er. Aliasinformationerne kan lettest repræsenteres med en graf som den for AnyExpression der ses i figur 5.9. Hver kasse repræsenterer en klasse, og pilene ud af en kasse repræsenterer de instansvariabler der ikke er primitive typer og peger på instansvariablens aliasrelationer. Grafen er i dette tilfælde cyklisk da en specialiseret multiplikation kan have deludtryk der også er specialiserede multiplikationer. Her er det vigtigt at forstå at hver kasse er en „repræsentation“ af instanser af de specialiserede objekter og at et objekt i programmet ikke nødvendigvist peger på sig selv. Som det ses i grafen, er der tre Variable-kasser, én for hver „**new** Variable()“ i analysekonteksten, og kun én Multiplication-kasse da der også kun er én „**new** Multiplication()“, nemlig når *inst_SpecMult* instansieres.

I semantikken for analysekonteksten har jeg ikke beskrevet reglerne for array-prædikater fordi der er mange faktorer i et array der kan specialiseres og derfor ville reglerne blive komplicerede. I AnyExpression er der et prædikat på arrayet *env* hvor arrayets længde er implicit statisk, og i konteksten i figur 5.7 ses det at *env* først tilordnes et nyt objekt af typen float[] med den statisk længde StaticValue.get_int(). Derefter tilordnes arrayets indhold bindingstider, og i dette tilfælde er der ingen restriktioner på indholdet der derfor er dynamisk. JSpec ved under analysen aldrig hvor mange elementer arrayet indeholder da der jo ikke er forskel på implicit og eksplicit statiske variabler. Derfor behandles alle indgange ens på en gang så at sige, og det gøres ved at sætte indgang nummer StaticValue.get_int() til den, i dette tilfælde, dynamiske værdi.

5.3 Specialiseringskontekst

I specialiseringskonteksten erklæres de eksakte værdier som programudsnittet skal specialiseres for. Hvis prædikaterne i specialiseringsklasserne er implicit statiske, læser JSpec de konkrete værdier fra værdifilen hvor brugeren har erklæret dem. I form ligner specialiseringskonteksten analysekonteksten; værdier skal tilordnes de variabler

```
public static void setSpecializationContext( ) {
   _this = meth_BinCube(0);
}
public static BinPower meth_BinCube(int n) {
   BinPower inst_BinCube = new BinPower( );
   String s_this_exp = "3";
   ObjectValue s_this_op = new ObjectValue("Mult", 0);
   inst_BinCube.exp = 3;
   inst_BinCube.op = new Mult( );
   return inst_BinCube;
}
```

Figur 5.10: Specialiseringskontekst til BinCube.

der bruges i prædikaterne. Dog skal der ikke gøres noget ved de dynamiske prædikater da de netop ikke har nogen faktisk værdi og ikke bruges under specialiseringen.

I dette kapitel vil jeg bruge de samme to eksempler som i det forrige til at illustrere specialiseringskonteksten.

Kontekseksempel til en binær eksponentialfunktion

Specialiseringskonteksten til BinCube fra figur 5.2 ses i figur 5.10, og den første del af semantikken for specialiseringskonteksten kan beses i figur 5.11. I semantikken for specialiseringskonteksten ses det at den starter på samme måde som analysekonteksten; med at erklære *_this* og parametrene. Men da den faktiske metode setSpecializationContext er en del af klassen _JSpec_Context, som også setAnalysisContext er, er instansvariablerne allerede definerede, og derfor ses de ikke i specialiseringskonteksten til BinCube. Specialiseringskonteksten og analysekonteksten har dog det fælles træk at de begge erklærer *inst_...* for hver specialiseringsklasse. I specialiseringskonteksten skal de faktiske værdier tilordnes hvor det i analysekonteksten gjaldt bindingstider.

På grund af de særlige egenskaber ved analysekonteksten hvor referencer til andre specialiseringsklasser refererer til samme objekt, har analysekonteksten den struktur der blev vist i forrige kapitel. Anderledes er det i specialiseringskonteksten hvor det er de faktiske værdier og klasser der skal tilordnes. Derfor kaldes hver tilordning af en specialiseringsklasse som en metode med et nummer som det ses i konteksten i figur 5.10 hvor *_this* tilordnes returværdien af meth_BinCube(0) hvor 0 er nummeret på specialiseringsklassen. Disse metodekalds funktion vil fremgå af næste eksempel om beregning af aritmetiske udtryk. I dette simple eksempel har de ingen betydning, og det er kroppen af metoden der er relevant for specialiseringskonteksten.

$[\![E\ \ S_1\ \ S_2\ \ \ldots\ \ S_m]\!] =$

public class SpecializationContext {
$[\![E]\!]\ \ [\![S\_1]\!] \ldots [\![S\_m]\!]$
}

$[\![$**specclass** E **specializes** $J\ \{p_1\ \ p_2\ \ \ldots p_k\ \ e\}]\!] =$

public static J *_this*;
$\mathcal{D}[\![e]\!]$
public static void setSpecializationContext( ) {
this = meth$E$(0);
}
public static J meth_E(int n) {
J *inst_E* = **new** J( );
$\mathcal{S}[\![p_1]\!](E,\epsilon)\ \ \mathcal{S}[\![p_2]\!](E,\epsilon) \ldots \mathcal{S}[\![p_k]\!](E,\epsilon)\ \ \mathcal{S}[\![e]\!](\epsilon,\epsilon)$
$\mathcal{A}[\![p_1]\!](E)\ \ \mathcal{A}[\![p_2]\!](E)\ \ \ldots\ \mathcal{A}[\![p_k]\!](E)\ \ \mathcal{A}[\![e]\!](\epsilon)$
return *inst_E*;
}

$[\![$**specclass** S_i **specializes** $J_i\{\ p_1\ \ p_2\ \ \ldots p_k\ \ e\ \}]\!] =$

public static J_i meth_S_i( ) {
J_i *inst_*S_i = new J_i( );
$\mathcal{S}[\![p_1]\!](S_i,\epsilon)\ \ \mathcal{S}[\![p_2]\!](S_i,\epsilon) \ldots \mathcal{S}[\![p_k]\!](S_i,\epsilon)$
$\mathcal{A}[\![p_1]\!](S_i)\ \ \mathcal{A}[\![p_2]\!](S_i) \ldots \mathcal{A}[\![p_k]\!](S_i)$
return *inst_*S_i;
}

$\mathcal{D}[\![$**public** int $f(T_1\ \ r_1,\ \ T_2\ \ r_2,\ \ \ldots, T_h\ \ r_h)\ \{$**where** $p_1\ \ \ldots$**where** $p_k\}]\!] =$

public static $T_1\ r_1$;
public static $T_2\ r_2$;
$\vdots$
public static $T_h\ r_h$;

Figur 5.11: Semantikstump for specialiseringskonteksten. Metoden setSpecializationContext erklæres. Desuden erklæres en metode for hver specialiseringsklasse hvori de faktiske værdier tilordnes.

$\mathcal{A}[\![v$ == $number]\!](C) =$
$\quad lhs(C,v)$ = $number$;
$\mathcal{A}[\![v$ == !$]\!](C) =$
$\quad lhs(C,v)$ = (Integer.valueOf)($str(C,v)$).intValue();
$\mathcal{A}[\![v : J]\!](C) =$
$\quad lhs(C,v)$ = **new** J();
$str(C,v) = s\_C\_v$
$str(\epsilon,v) = s\_v$

Figur 5.12: Semantikstump for specialiseringskonteksten. Tilordningen af de faktiske værdier for „simple“ prædikater.

Jeg vil begynde nedefra i metoden hvor de faktiske værdier tilordnes og da *exp* i BinCube er eksplicit statisk med værdien 3, tilordnes 3 til *inst_BinCube.exp*. Den eksplicit statiske typeerklæring af *op* tilordnes et nyt objekt af typen Mult. I figur 5.12 ses semantikken for disse simple prædikater.

De tilordninger der foretages i metoden før tilordningen af de faktiske værdier, bruges til generering af vogteren som jeg vil beskrive i kapitel 5.4, og til at tilordne implicit statiske værdier læst fra værdifilen. I dette eksempel er der ingen implicit statiske prædikater og instansvariablerne bruges ikke, men det gør de i næste eksempel hvor jeg vil gennemgå semantikken og brugen af disse tilordninger.

Kontekseksempel til beregning af aritmetiske udtryk

Specialiseringsklassen AnyExpression fra figur 5.6 indholder referencer til andre specialiseringsklasser, rekursive erklæringer og implicit statiske prædikater hvilket dækker de regler i semantikken for specialiseringskonteksten jeg endnu ikke har gennemgået. Specialiseringskonteksten for AnyExpression ses i figur 5.13. For at få plads til konteksten har jeg reduceret tilordningerne af *inst_SpecMult.left* og *right* som er ækvivalente med tilordningerne af *e*.

Jeg vil begynde med at gennemgå det forsømte fra forrige eksempel, nemlig tilordninger af de faktiske værdier fra de implicit statiske variabler. Semantikstumpen for disse ses i figur 5.14 og i bunden af figuren ses hjælpefunktionen *vn* (*variable name*). Figur 5.15 er de faktiske værdier til AnyExpression der som eksemplet i kapitel 4.8 specialiserer for at beregne arealet af en cirkel. Det er fra denne fil at setSpecializationContext skal læse værdierne. Den første værdi der læses fra konteksten til AnyExpression, er typen på parameteren *e* og det foregår via en metode getTypeValue med den aktuelle specialiseringsklasse, navnet på instansvariablen (her med metodenavnet da det er

```
public static void setSpecializationContext() {
  _this = meth_AnyExpression(0);
}
public static Calculate meth_AnyExpression(int n) {
  Calculate inst_AnyExpression = new Calculate();
  ObjectValue s_e = getTypeValue("AnyExpression", "calc(e)", n);
  ObjectValue s_env = getArray("AnyExpression", "calc(env)", "float",null, new String[]
{}, n);
  if (s_e.value.equals("SpecMult")) {
    e = meth_SpecMult(s_e.n);
  } else {
    if (s_e.value.equals("SpecConst")) {
      e = meth_SpecConst(s_e.n);
    } else {
      e = new Variable();
    }
  }
  env = new float[s_env.lengths[0]];
  return inst_AnyExpression;
}
public static Multiplication meth_SpecMult(int n) {
  Multiplication inst_SpecMult = new Multiplication();
  ObjectValue s_this_left = getTypeValue("SpecMult", "left", n);
  ObjectValue s_this_right = getTypeValue("SpecMult", "right", n);
  guard_SpecMult(s_this_left, s_this_right, n);
  if (s_this_left.value.equals("SpecMult")) {
    inst_SpecMult.left = meth_SpecMult(s_this_left.n);
  } else {
    ...
  }
  if (s_this_right.value.equals("SpecMult")) {
    inst_SpecMult.right= meth_SpecMult(s_this_right.n);
  } else {
    ...
  }
  return inst_SpecMult;
}
public static Constant meth_SpecConst(int n) {
  Constant inst_SpecConst = new Constant();
  String s_this_value = getValue("SpecConst", "value", n);
  inst_SpecConst.value = (Float.valueOf(s_this_value)).floatValue();
  return inst_SpecConst;
}
```

Figur 5.13: Specialiseringskontekst til AnyExpression.

$\mathcal{S}[\![v \texttt{ == } number]\!](C,m) =$
String $str(C,v)$ = $number$;
$\mathcal{S}[\![v \texttt{ == !}]\!](C,m) =$
String $str(C,v)$ = getValue("C", "$vn(m,v)$", n);
$\mathcal{S}[\![v\texttt{:}J]\!](C,m) =$
ObjectValue $str(C,v)$ = **new** ObjectValue("J", n);
$\mathcal{S}[\![v\texttt{:}S]\!](C,m) =$
ObjectValue $str(C,v)$ = getTypeValue("C", "$vn(m,v)$", n);

$\mathcal{S}[\![v\texttt{: } T_1\texttt{|}T_2\texttt{|...|}T_k\texttt{;}]\!](C,m) =$
ObjectValue $str(C,v)$ = getTypeValue("C", "$vn(m,v)$", n);

$\mathcal{S}[\![\textbf{public int } f\texttt{(}T_1\ r_1\texttt{, }T_2\ r_2\texttt{, ...,}T_l\ r_l\texttt{) \{}\textbf{where } p_1 \texttt{ ...}\textbf{where } p_k \texttt{ \}}]\!](C,m) =$
$\mathcal{S}[\![p_1]\!](C,f)\ \mathcal{S}[\![p_2]\!](C,f)\ \ldots\ \mathcal{S}[\![p_k]\!](C,f)$
$vn(m,v) = m(v)$
$vn(\epsilon,v) = v$

Figur 5.14: Semantikstump til specialiseringskonteksten. Indsamling af de faktiske værdier.

en parameter) plus et heltal n som argumenter. Variablen n er nummeret på den specialiseringsklasse der skal læses i og da AnyExpression ikke har noget nummer, er n i dette tilfælde 0. Metoden getTypeValue returnerer flere værdier; den faktiske værdi som i dette tilfælde er „SpecMult“, jævnfør figur 5.15, og det nummer der eventuelt står efter #. I dette tilfælde står der ikke noget, og 0 returneres. Disse værdier gemmes i en lokal variabel der har typen ObjectValue. Noget tilsvarende sker i næste linje af konteksten hvor de faktiske værdier til arrayet *env* skal læses. Som nævnt tidligere er der mange faktorer der kan specialiseres med hensyn til ved arrays og for at få de korrekte oplysninger fra filen, bruges metoden getArray der som argumenter tager alt det der på forhånd er kendt i arrayet i følgende rækkefølge: navnet på specialiseringsklassen, navnet på variablen, typen på arrayet, længden af arrayet (muligvis flerdimensionelt), indholdet af arrayet samt nummeret på specialiseringsklassen. De oplysninger der skal læses fra filen, gives med metoden som **null**, og i eksemplet er det længden der skal læses, så på længdens plads står „**null**“. Indholdet af arrayet er dynamisk, og skal ikke læses, og derfor sendes et tomt array af strenge med. Metoden returnerer alle oplysningerne om arrayet til *s_env*, både dem der blev givet som argument og dem der er læst fra værdifilen, og oplysningerne kan herefter bruges til videre specialisering. I tilfælde hvor variablerne er eksplicit statiske som i BinCube, tilordnes værdierne direkte som

```
AnyExpression {
   calc(e) : SpecMult;
   calc(env) : float[1];
}
SpecMult {
   left : SpecMult#1;
   right : SpecConst;
}
SpecMult#1 {
   left : Variable;
   right : Variable;
}
SpecConst {
   value = 3.1416F;
}
```

Figur 5.15: Værdifil til beregning af arealet af en cirkel for AnyExpression.

$$\mathcal{A}[\![v : S]\!](C) =$$
$$\quad lhs(C,v) \texttt{ = meth\_}S\texttt{(}str(C,v).n\texttt{);}$$
$$\mathcal{A}[\![v : \ T_1 \texttt{|} T_2 \texttt{|} \ldots \texttt{|} T_n\texttt{;}]\!](C) =$$
$$\quad \textbf{if (}str(C,v).s\texttt{.equals(``}T_1\texttt{'')) \{}$$
$$\quad\quad \mathcal{A}[\![v : T_1]\!](C) \texttt{ ;}$$
$$\quad \textbf{\} else \{}$$
$$\quad\quad \mathcal{A}[\![v : T_2 \texttt{|} \ldots \texttt{|} T_n]\!](C)$$
$$\quad \texttt{\}}$$
$$\mathcal{A}[\![\textbf{public int } f \texttt{ (}T_1 \ r_1\texttt{,}T_2 \ r_2\texttt{, . . . , } T_h \ r_h\texttt{)\{}\textbf{where } p_1 \texttt{. . . } \textbf{where } p_k\texttt{\}}]\!](C) =$$
$$\quad \mathcal{A}[\![p_1]\!](\epsilon) \ \mathcal{A}[\![p_2]\!](\epsilon) \ldots \mathcal{A}[\![p_k]\!](\epsilon)$$
$$str(C,v) = s\_C\_v$$
$$str(\epsilon,v) = s\_v$$

Figur 5.16: Semantikstump for specialiseringskonteksten. Tilordning af de faktiske værdier for „avancerede“ prædikater.

det netop sås i eksemplet. Udover de to metoder jeg her har nævnt, bruges metoden getValue til at læse en primitiv værdi som det ses nederst i specialiseringskonteksten i metoden meth_SpecConst. Her kan værdien opbevares i en streng da der ikke er flere versioner som ved specialiseringsklasser, og når værdien skal bruges skal float-værdien trækkes ud af strengen. Jeg antager her at det er korrekte værdier der står i værdifilen under antagelse af at der værktøj der skal bruges til at modificere filen kun lader tilladte værdier indtaste.

Semantikken for resten af reglerne for tilordning af de faktiske værdier ses i figur 5.16. Jeg vil begynde med den anden regel i semantikken; den der beskriver implicit statiske typeerklæringer. Et eksempel på reglen ses i analysekonteksten for AnyExpression, blandt andet for e – her bruges at s_e har en instansvariabel, *value*, der indeholder den faktiske værdi. Fra specialiseringsklassen kendes de mulige klasser og ved at sammenligne tekststrenge findes den rigtige tilordning som i dette tilfælde er SpecMult der som

bekendt er en specialiseringsklasse, og derfor skal alle de specialiserede værdier fra SpecMult også tilordnes *e*, og det gøres ved at tilordne *e* returværdien af meth_SpecMult som netop returnerer et objekt af typen Multiplication med alle de faktiske værdier fra SpecMult. Metoden meth_SpecMult kaldes med nummeret på specialiseringsklassen som i dette tilfælde er nul, det vil sige at specialiseringsklassen ikke har noget nummer (jævnfør figur 5.15). I metoden meth_SpecMult læses på samme måde de faktiske værdier fra værdifilen, og *s_this_left* vil indeholde værdierne SpecMult og 1 og når der derefter kaldes videre gennem meth_SpecMult, er det med argumentet 1, og det vil dermed blive de korrekte værdier der læses fra værdifilen. Med disse metodekald sikrer jeg at det kun er de specialiseringsklasser der rent faktisk nås, der bliver læst, og det er derfor uden betydning hvad der står i resten. I analysekonteksten var der kun én instans af Multiplication som blev brugt af JSpec som repræsentant for alle instanser. I specialiseringskonteksten er der en instans af Multiplication for hver gang klassen bruges. Da alle metodekaldene starter i *_this*, indholder *_this* nu alle de specialiserede værdier som JSpec skal bruge til at generere den specialiserede kode.

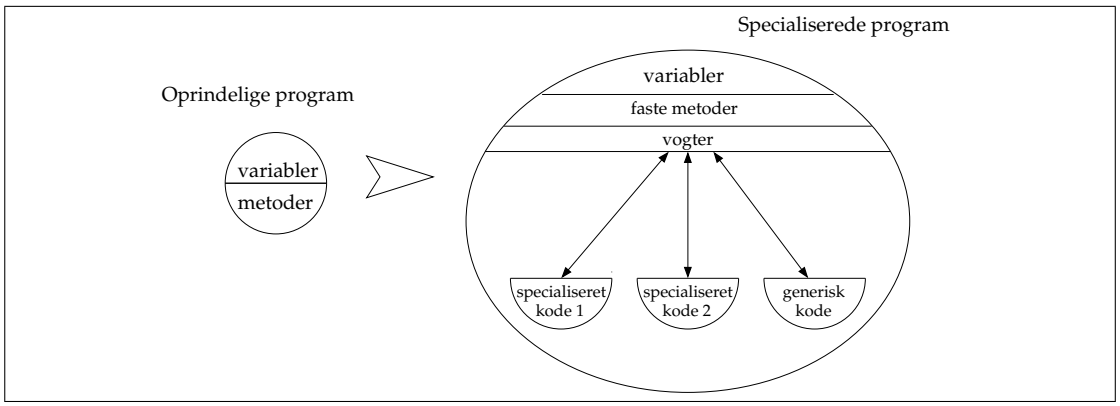
Den specialiserede kode som JSpec genererer ud fra analyse- og specialiseringskonteksten, gemmes i en fil der senere sammen med vogteren skal sættes ind i det oprindelige program af AspectJ.

5.4 Vogtere

En vogter kontrollerer om betingelserne for at kalde den specialiserede kode er opfyldte inde koden kaldes. JSpec sørger for at betingelserne er opfyldt i hele det specialiserede kode – hvis ikke bliver koden ikke specialiseret. Det er specielt vogterne der adskiller Pesto fra tidligere brug af JSpec hvor programmøren selv skulle afgøre hvornår den specialiserede kode skulle kaldes.

Vogteren kan genereres når de statiske værdier er kendt, og det gøres i setSpecializationContext samtidig med at de specialiserede værdier tilordnes.

Hver specialisering har én vogter som kaldes inden udførelsen af den metode programmet er specialiseret ud fra, det vil sige indgangspunktet. Betingelserne for at kalde den specialiserede kode er at de værdier koden er specialiseret for er de faktiske værdier når den metode der er specialiseret kaldes.



Figur 5.17: Grafisk fremstilling af transformationen af et program til at indeholde vogtere og flere versioner af programudsnittet.

Som set tidligere, ønskes en betingelse rundt om indgangsmetoden i retningen af:

```
if (betingelser for specialiseret kode 1) {
   specialiseret metode #1;
} else {
   if (betingelser for specialiseret kode 2) {
      specialiseret metode #2;
   } else {
      oprindelige metode;
   }
}
```
(25)

Dette gøres i praksis ved at bruge aspektorienteret programmering (se kapitel 2.5) og AspectJ som netop kan finde et bestemt punkt i koden og indsætte et givet stykke kode på dette punkt netop som jeg har brug for.

Figur 5.17 viser en grafisk fremstilling af det specialiserede program hvor det oprindelige program er vist som en cirkel til venstre for pilen, og det specialiserede program til højre. Det specialiserede program er delt op i variabler, faste metoder, vogter og specialiserede metoder. Variablerne ændres ikke i den specialiserede kode, og det gør de metoder der ikke er berørt af specialisereringen naturligvis heller ikke. Figuren viser at i stedet for det oprindelige indgangspunkt er der i det specialiserede program en vogter, en eller flere specialiserede metoder samt den oprindelige metode. Pilene fra vogteren er ikke referencer, men blot en illustration af at vogteren sørger for at kalde det rigtige kode på det rigtige tidpunkt.

$[\![E\ \ S_1\ \ S_2\ \ \ldots\ \ S_m]\!] =$
$\quad [\![E]\!]\ [\![S\_1]\!] \ldots [\![S\_m]\!]$
$[\![$**specclass** E **specializes** $J\{p_1\ \ p_2\ \ \ldots p_k\ \ e\}]\!] =$
$\quad$ **public aspect** $E\{$
$\quad\quad$ **public** boolean J.guard_E($\mathcal{P}[\![e]\!]$) {
$\quad\quad\quad \mathcal{G}[\![p_1]\!]$(this.) $\mathcal{G}[\![p_2]\!]$(this.) $\ldots$ $\mathcal{G}[\![p_k]\!]$(this.) $\mathcal{G}[\![e]\!](\epsilon)$
$\quad\quad\quad$ **return** true;
$\quad\quad$ }
$\quad\quad$ **pointcut** entrypoint ($J\_j, \mathcal{P}[\![e]\!]$):
$\quad\quad\quad \mathcal{C}[\![e]\!](J)$;
$\quad\quad \mathcal{O}[\![e]\!](J, E)$
$\quad$ }
$[\![$**specclass** S_i **specializes** $J_i\{p_1\ \ p_2\ \ \ldots p_k\ \ e\}]\!] =$
$\quad$ **public** boolean $J_i i$._guard_S_i( ){
$\quad\quad \mathcal{G}[\![p_1]\!]$(this.) $\mathcal{G}[\![p_2]\!]$(this.) $\ldots \mathcal{G}[\![p_k]\!]$(this.)
$\quad\quad$ **return** true;
$\quad$ }
$\mathcal{P}[\![$**public** int f($T_1\ \ r_1$, $\ T_2\ \ r_2$, $\ \ldots, T_h\ \ r_h$) { **where** $p_1\ \ \ldots$ **where** $p_k\}]\!] =$
$\quad T_1\ r_1, T_2\ r_2, \ldots, T_h\ r_h$

Figur 5.18: Semantikstump fra vogteren. Erklæring af aspektet plus pointcut og erklæring af vogtermetoder for hver specialiseringsklasse.

Vogter til en binær eksponentialfunktion

Vogterkoden genereres altså af setSpecializationContext samtidig med tilordningen af de faktiske værdier. I de semantikstumper jeg har vist fra specialiseringskonteksten i det forrige kapitel, har jeg pillet alt der har haft med vogteren at gøre ud for at reglerne skulle blive lettere at overskue. I dette kapitel har jeg ikke blot klippet de semantikregler der ikke genererer vogtere ud, jeg har også ændret det kode i reglerne der genererer vogtere, så det kode der står i semantikken er det der genereres af setSpecializationContext når specialisering udføres. Hvis det skulle være helt korrekt, skulle der rundt om hver genererede linje i reglerne stå „guardLine("...')';" da guardAppend er den metode i _JSpec_Context der skriver en linje vogterkode i en fil, men det ville ikke hjælpe til forståelsen af koden så det har jeg udeladt. Den fulde semantik for specialiseringskonteksten samt et eksempel på en kontekst ses i bilag C og D. I figur 5.18 ses den del af semantikken der for hver specialiseringsklasse opretter en vogter og en gang for alle opretter pointcuttet.

Figur 5.19 viser vogteren for BinCube fra figur 5.2 skrevet i AspectJ der har Java som basissprog med nogle ekstra nøgleord og konstruktioner. I første linje erklæres aspektet,

```
public aspect BinCube{

  private boolean BinPower._guard_BinCube(int base){
    if (!(this.exp == 3)) return false;
    if (!(this.op.getClass() == Mult.class)) return false;
    return true;
  }

  pointcut entrypoint (BinPower _binpower, int _raise_base):
  call (int BinPower.raise(int))
    && args(_raise_base)
    && target(_binpower);

  int around(BinPower _binpower, int _raise_base):
  _raise_entrypoint(_binpower, _raise_base) {
    if (_binpower._guard_BinCube(_raise_base)) {
      return _binpower._raise_spec(_raise_base);
    } else {
      return proceed(_binpower, _raise_base);
    }
  }
}
```

Figur 5.19: Vogterkoden til BinCube.

$$\mathcal{G}[\![v \texttt{ == } number]\!](p) =$$
if (!(*pv* == *number*)) **return** false;
$$\mathcal{G}[\![v : J]\!](p) =$$
if (!(*pv*.getClass() == *J*.**class**)) **return** false;

Figur 5.20: Semantikstump for vogterkoden. Vogtere for de enkelte prædikater erklæres.

og den første metode guard_BinCube introduceres,[2] det vil sige indsættes, i BinPower-klassen, og har derefter udsyn til andre metoder i klassen. Metoden er klassens vogter som kontrollerer at værdierne er korrekte i forhold til specialiseringen hvilket i eksemplet vil sige at kontrollere at *exp* er 3 og at *op* har typen Mult og returnere „true“ hvis betingelserne er opfyldt, og ellers „false“. Hvis der havde været et prædikat på parameteren *base*, skulle den selvfølgelig også kontrolleres, og derfor gives parameteren med, og for at gøre det lidt nemt gives alle indgangspunktets parametre med som argument til vogteren. Vogteren er *private*, og kan derfor ikke bruges af andre end klassen selv.

[2] *introduktion* er det ord der bruges i AspectJ-dokumentationen. I en tidligere version af AspectJ brugtes nøgleordet **introduce**, men i denne version (1.0.6) introduceres metoder og felter ved blot at erklære dem med klassenavnet.

$\mathcal{C}[\![$**public** int $f($**(**T_1 r_1**,** T_2 r_2**,** ... **,** T_h r_h**)** **{where** p_1 ... **where** p_k**}**$]\!](R) =$
call (int $R.f(T_1, T_2, \ldots, T_h)$)
&& **args**$(\_r_1, \_r_2, \ldots, \_r_h)$
&& **target**$(\_r)$;

Figur 5.21: Semantikstump for pointcut i vogterkoden.

$\mathcal{O}[\![$**public** int $f($**(**T_1 r_1**,** T_2 r_2**,** ... **,** T_h r_h**)** **{where** p_1 ... **where** p_k**}**$]\!](R, L) =$
int **around**$(R\ \_r, T_1\ \_r_1, T_2\ \_r_2, \ldots, T_h\ \_r_h)$:
entrypoint$(_r, _r_1, _r_2, \ldots, _r_h)$ {
if $(\_r$.guard_$L(_r_1, _r_2, \ldots, _r_h)$
return $_j._f$_spec$(_r_1, _r_2, \ldots, _r_h)$;
} **else** {
return proceed$(\_R, \_r_1, \_r_2, \ldots, \_r_h)$;
}
}
hvor $r = lowercase(R)$ og $j = lowercase(J)$

Figur 5.22: Semantikstump for around i vogterkoden.

Semantikstumpen for genereringen af vogtere for de simple prædikater ses i figur 5.20.

Den næste blok i koden er en **pointcut**-funktion. Pointcut er en konstruktion i AspectJ der tager fat i et givet punkt i metoden. Det kan blandt andet være hvor en givet variabel bruges eller som her hvor en metode kaldes. Den metode vi ønsker en pegepind til, er BinCubes indgangspunkt, raise, og pointcuttet her hedder derfor entrypoint. Desuden ønsker vi at kunne tilgå både klassen omkring metoden (BinPower) og metodens argumenter (*base*) for at kontrollere værdierne og det der sker i kroppen på pointcut ved brug af **args** og **target** og gemme dem i henholdsvis *_raise_base* og *_binpower*. Semantikken for pointcuttets krop afhænger kun af indgangspunktet som det ses i figur 5.21.

Den sidste blok er **around**-funktionen som *rundt om* et givet pointcut (her entrypoint) sætter around-funktionens krop. Det er her konstruktionen fra kodestump 25 bruges. Hvis betingelserne er opfyldt, det vil sige at guard_SpecPower returnerer „true“, kaldes den specialiserede kode der også er indsat i BinPower-klassen ved at bruge AspectJ. Hvis betingelserne ikke er opfyldt, kaldes **proceed**, det vil sige den metode som aspektet ligger rundt om, altså den generiske metode, raise. I dette tilfælde returnerer **around** et heltal da raise returnerer et heltal. Semantikken for around, der som den for pointcut også kun afhænger af indgangspunktet, ses i figur 5.22.

```
public aspect Area{
  private boolean Calculate._guard_AnyExpression(Exp e, float[] env){
    if (!(e.getClass() == Multiplication.class)) return false;
    if (!(((Multiplication)e)._guard_SpecMult0())) return false;
    if (!(env.getClass() == float[].class)) return false;
    if (!(env.length == 1)) return false;
    return true;
  }
  private boolean Multiplication._guard_SpecMult0(){
    if (!(this.left.getClass() == Multiplication.class)) return false;
    if (!(((Multiplication)this.left)._guard_SpecMult1())) return false;
    if (!(this.right.getClass() == Constant.class)) return false;
    if (!(((Constant)this.right)._guard_SpecConst0())) return false;
    return true;
  }
  private boolean Multiplication._guard_SpecMult1(){
    if (!(this.left.getClass() == Variable.class)) return false;
    if (!(this.right.getClass() == Variable.class)) return false;
    return true;
  }
  private boolean Constant._guard_SpecConst0(){
    if (!(this.value == 3.1416F)) return false;
    return true;
  }
  pointcut _calc_entrypoint (Calculate _calculate, Exp _calc_e, float[] _calc_env):
  call (float Calculate.calc(Exp, float[]))
    && args(_calc_e, _calc_env)
    && target(_calculate);
  float around(Calculate _calculate, Exp _calc_e, float[] _calc_env):
  _calc_entrypoint(_calculate, _calc_e, _calc_env) {
    if (_calculate._guard_AnyExpression(_calc_e, _calc_env)) {
      return _calculate._calc_spec(_calc_e, _calc_env);
    } else {
      return proceed(_calculate, _calc_e, _calc_env);
    }
  }
}
```

Figur 5.23: Vogteraspekt til AnyExpression.

$\mathcal{G}[\![v : S]\!](p) =$
if (!(*pv*.getClass() == R_S.*class*)) **return** false;
if (!(((*R*$_S$)*pv*)._guard_*S*())) **return** false;
hvor $R_S = rootclass(S)$
$\mathcal{G}[\![$**public** int *f*(*T*$_1$ *r*$_1$, *T*$_2$ *r*$_2$, ..., *T*$_h$ *r*$_h$) { **where** *p*$_1$ ... **where** *p*$_l$}$]\!](p) =$
$\mathcal{G}[\![p_1]\!](p)\ \mathcal{G}[\![p_2]\!](p)\ \ldots \mathcal{G}[\![p_k]\!](p)$

Figur 5.24: Semantikstump for vogterkoden. Generering af vogtere.

Vogter til beregning af aritmetiske udtryk

Figur 5.23 viser vogteren for specialiseringsklassen AnyExpression fra figur 5.6. Her ses det at der er en vogter for hver specialiseringsklasse, og som i specialiseringskonteksten kaldes der mellem disse vogter-metoder når der refereres mellem specialiseringsklasser. Vogterne har også et nummer som er det nummer den tilhørende specialiseringsklasse har (og 0 for intet nummer). Resten af semantikken for vogterne ses i figur 5.24. Når en typeerklæring refererer til en specialiseringsklasse, kontrolleres naturligvis først at instansvariablen har den rette type som ved en almindelig typeerklæring. Derefter kan instansvariablen *castes* til den type den netop er testet for at have, og vogtermetoden, der jo også er introduceret i rodklassen af aspektet, kaldes. Under genereringen af vogteren fra setSpecializationContext bliver de faktiske værdier, der eventuelt er hentet i værdifilen, brugt til at generere vogtermetoderne. Læg mærke til at der ikke er nogle regler for implicit statiske prædikater da alle værdier netop er kendte når vogterne genereres.

Pointcut- og aroundmetoderne for dette eksempel svarer i det store hele til dem for BinCube.

Placering af vogteren

Som antydet tidligere i specialet er det afgørende for specialiseringen at programmøren overvejer hvor han sætter sin vogter da selve kaldet til vogteren og de sammenligninger der er i vogteren tager tid og det derfor er bedst at få vogteren udført så sjældent som muligt. For at vise vigtigheden i placering af vogteren har jeg sammenlignet kørsler af hver af de to eksempelprogrammer fra dette kapitel hvor vogteren er placeret henholdsvis inden for og uden for den kritiske kode. Eksperimenterne er foretaget efter samme skabelon som tidligere eksperimenter omtalt i specialet.

At placere indgangspunktet udenfor den kritiske kode vil sige at sørge for at indgangspunktet ikke er en metode der bliver kaldt ved hver eneste iteration af programmet. For BinPower-eksempler vil det sige at hvis indgangspunktet er raise som i BinCube og

	Tid				
Virtuel maskine	**Oprindeligt program**	**Vogter inden for**	**Forbedring**	**Vogter uden for**	**Forbedring**
Hotspot 1.3	1586	2369	-49%	1845	-16%
-server	1027	1134	-10%	167	515%
Hotspot 1.4	2728	1554	76%	378	622%
-server	1343	319	321%	168	700%
IBM JIT 1.3.1	747	653	14%	168	345%

Figur 5.25: Forskel på at placere indgangspunktet inden for og uden for den kritiske kode i BinPower.

en løkke sættes til at kalde raise 1000 gange så vil vogteren for raise blive kaldt 1000 gange. Det er der i dette tilfælde ingen grund til. I stedet har jeg konstrueret en klasse med en metode der indeholder en løkke der kalder raise, og denne nye metode lader jeg være indgangspunkt for specialiseringen. Derved har jeg fjernet indgangspunktet og dermed vogteren fra det kritiske kode.

Tabellen i figur 5.25 viser resultaterne af kørslen, og ikke overraskende er der en klar fordel ved at vogteren placeres uden for den kritiske kode. Hvad der dog er interessant, om ikke ligefrem overraskende, er at det med nogle virtuelle maskiner rent faktisk stadig kan betale sig at specialisere selv om vogteren bliver kaldt hver eneste gang den kritiske kode kaldes. Faktisk giver det i et enkelt tilfælde en god forbedring på 3,2 gange den oprindelige hastighed. Grunden til at dette er et interessant resultat er at det måske ikke altid er muligt at fjerne indgangspunktet fra den kritiske kode, og resultatet viser at det i nogle tilfælde godt kan betale sig at specialisere ved at bruge Pesto alligevel hvilket bestemt ikke var min antagelse før jeg gennemførte eksperimentet.

I figur 5.26 ses de tilsvarende målinger for beregning af aritmetiske udtryk. Når programmet specialiseres bliver hele udtrykket statisk, det vil sige det bliver specialiseret bort. Derfor er forbedringerne ved at køre det specialiserede program meget gode for dette program hvis vogteren bliver placeret hensynmæssigt. I sidste kolonne ses at programmet kører 15,7 gange hurtigere med det specialiserede kode end med det oprindelig køre på IBMs virtuelle maskine som også er den der udfører det specialiserede program hurtigst (og desuden også det oprindelige program). Men som det fremgår af ovenstående kapitler er vogteren for AnyExpression mere kompliceret end den for BinCube da den kontrollerer hele strukturen af det udtryk indgangspunktet kaldes med og som det ses i kolonnen med forbedringer ved kørsel af det specialiserede program med

	Tid				
Virtuel maskine	**Oprindeligt program**	**Vogter inden for**	**Forbedring**	**Vogter uden for**	**Forbedring**
Hotspot 1.3	1616	13305	-723%	902	79%
-server	1265	9305	-636%	90	1300%
Hotspot 1.4	1906	1276	49%	560	277%
-server	1259	1077	17%	506	149%
IBM JIT 1.3.1	1036	653	-3%	62	1571%

Figur 5.26: Forskel på at placere indgangspunktet indenfor og udenfor den kritiske kode i udtrykseksemplet.

```
specclass KnownOp specializes BinPower {
  exp == 3;
  op : Mult | Add;
  public int raise(int base);
}
```

Figur 5.27: Specialiseringsklassen KnownOp.

vogteren inden for, mister man meget ved at placere en beregningstung vogter forkert. Der er dog en rimelig forbedring på 49% med Hotspot 1.4, men set i forhold til hvor meget programmet er specialiseret er det uacceptabelt.

Flere specialiseringer af et program

Hvis en specialiseringsklasse ikke har nogle implicit statiske variabler der skal læses fra en værdifil som for eksempel BinCube, opkaldes aspektet efter indgangspunktsspecialiseringsklasssen. Hvis der derimod er en værdifil som specialiseringsklassen køres med, opkaldes aspektet efter navnet på værdifilen som i figur 5.23 hvor aspektet hedder Area efter værdifilen *Area.values*. Dermed er det muligt at have forskellige vogtere til forskellige værdifiler uden at de overskriver hinanden.

Figur 5.27 er en implicit statisk version af BinCube hvor eksponenten stadig er tre, men funktionens basisoperation enten kan være Mult eller Add. Til denne specialisering er der to forskellige værdifiler, *AddOp.values* (figur 5.28 (a)) og *MultOp.values* (figur 5.28 (b)), og for hver værdifil bliver der genereret en vogter, *AddOp.java* og *MultOp.java*, som indeholder vogterkode ækvivalent til den i figur 5.19. Hvis man forsøger at indsætte en metode i en klasse som allerede har en metode med samme navn, giver AspectJ en fejl – med mindre metoden er privat da private metoder jo kun kan kaldes fra andre

KnownOp {
exp = 3;
op : Add;
}
	KnownOp {
	exp = 3;
	op : Mult;
	}

(a) *AddOp.values*	(b) *MultOp.values*

Figur 5.28: Værdifiler til KnownOp.

metoder indsat af aspektet. Vogterkoden er privat, og det er derfor muligt at sætte to vogtermetoder med samme navn ind i samme klassefil hvilket vil være tilfældet hvis man specialiserer samme program med forskellige værdifiler – AspectJ sørger for at den rigtige vogter bliver kaldt.

For at kunne bruge flere forskellige versioner af specialiseret kode i ét program kaldes AspectJ separat efter de respektive specialiseringer af koden er foretaget med de respektive specialiseringer. Vogterkoden vil ligne den i kodeeksempel 25:

```
if (_guard_BinCube_Add(_raise_base)) {
  return _binpower._raise_spec_Add(_raise_base);
} else {
  if (_guard_BinCube_Mult(_raise_base)) {
    return _binpower._raise_spec_Mult(_raise_base);
  } else {
    return proceed(_binpower, _raise_base);
  }
}
```
(26)

Der vil dermed være flere betingelser der skal kontrolleres hver gang metoden kaldes og koden bliver derved langsommere. Det burde fremgå af dette kapitel at jo flere prædikater der sættes i et program, jo flere ting skal kontrolleres, og jo længere tager det, hvilket der skal tages højde for når specialiseringsklasserne skrives. Som nævnt er det vigtigt at tage højde for strukturen i programmet når der specialiseres. Hvis indgangspunktet ofte kaldes vil vogteren blive udført hver gang, og i det tilfælde kan det måske betale sig at lægge en metode uden om det punkt i sin kode og lade kroppen i denne metode kalde det punkt der ønskes specialiseret og lade den nye metode være specialiseringens indgangspunkt så vogterkoden dermed kun bliver kørt én gang.

Man kan sagtens forestille sig andre måder at implementere en vogter på end den jeg har valgt i denne version af Pesto hvilket vil blive diskuteret i kapitel 6.2.

Eksperimenter

Som beskrevet, har det tidligere i JSpec været nødvendigt for programmøren selv at sørge for at kalde det specialiserede kode på det rigtige tidspunkt. Med indførelsen af

	Tid					
Virtuel maskine	**Oprindeligt program**	**Specialiseret uden vogter**		**Specialiseret med vogter**		**Tab ved brug af vogter**
Hotspot 1.3	7977	4849	65%	9118	-14%	89%
-server	5102	840	507%	837	509%	0%
Hotspot 1.4	13652	1914	613%	1879	627%	-2%
-server	6715	843	697%	833	706%	- 1%
IBM JIT 1.3.1	3778	847	346%	842	349%	0%

Figur 5.29: Kørsel af BinCube med og uden vogter genereret af Pesto.

vogtere er det ikke længere nødvendigt, men et relevant spørgsmål er hvor meget overhead det skaber at have vogterkode. For at undersøge dette har jeg foretaget nogle eksperimenter og sammenlignet kode for specialiserede programmer med og uden vogtere genereret af Pesto. Det drejer sig om de to eksempler jeg har kigget på i dette kapitel, nemlig BinCube og AnyExpression. Målingerne er foretaget på samme basis som tidligere målinger i dette speciale.

Figur 5.29 viser resultaterne fra BinCube. Den første kolonne er programmets udførselstid med den givne vituelle maskine på 10 mio. omgange. Som det ses, er der stor forskel, og det er IBMs virtuelle maskine der har den bedste udførselstid på programmet. Den anden kolonne viser udførselstiden for den specialiserede kode uden vogter og kolonne tre viser forbedringen uden vogter. I praksis betyder tallene at Hotspot 1.3 -server, Hotspot 1.4 -server og IBM udfører programmet lige hurtigt. Fjerde og femte kolonne viser udførselstiden og forbedringen med vogter – vogteren er placeret optimalt, det vil sige uden for det kritiske kode. Her ses det at vogteren giver så stort et overhead at programmet faktisk er langsommere end det oprindelige program kørt med Hotspot 1.3. Det vil sige at programmet kører 89% langsommere med den specialiserede kode med vogter end uden som det ses i fjerde kolonne. Dette er meget overraskende og hvorfor dette overhead opstår, har jeg ikke noget bud på. Det der tilføjes i programmet, er en vogter der kontrollerer at eksponenten har den rette værdi og at operationen er af den rette type. I resten af kolonne fire ses det at der ikke er noget tab ved at bruge vogteren. Faktisk kører koden en smule hurtigere hvilket jeg heller ikke kan forklare – jeg har foretaget gentagne test for at være sikker på at det ikke var tilfældigt.

Anderledes ser det ud i figur 5.30. Kolonnerne er arrangeret på samme måde som i figur 5.29, og her ses det at Hotspot 1.4 -server er hurtigste virtuelle maskine på det op-

	Tid					
Virtuel maskine	**Oprindeligt program**	**Specialiseret uden vogter**		**Specialiseret med vogter**		**Tab ved brug af vogter**
Hotspot 1.3	6821	2799	144%	4247	61%	52%
-server	5866	384	1420%	499	1075%	30%
Hotspot 1.4	6904	966	615%	1637	322%	70%
-server	3535	159	2100%	809	373%	400%
IBM JIT 1.3.1	4697	294	1490%	305	1242%	19%

Figur 5.30: Kørsel af AnyExpression med og uden vogter genereret af Pesto.

```
#!/usr/local/bin/tclsh
# This file is generated automaticly by Pesto
# Jspec config file: pesto_AnyExpression_mdl.jspec
= module "pesto_AnyExpression_mdl"
= entry_point {float Calculate.calc(Exp e, float[] env)}
= entry_point_signature {S S S}
= specialized_entry_point "_calc_spec"
= main_classes {Negation Addition Variable Calculate Binary Multiplication Unary Constant Exp}
= compile_during_weave 0
= heap_size 200000
= new_aspectj_syntax 1
```

Figur 5.31: Modulfilen til Calculate-eksemplet.

rindelige kode hvilket også gør sig gældende for det specialiserede kode uden vogter. For koden med vogter er det IBMs JIT der er hurtigst. Dermed er det også med IBMs der er det mindste tab når koden udføres med vogter. Som det ses, er der større tab ved at bruge vogtere i dette eksempel end i forrige. Det kan hovedsageligt forklares med at vogteren i dette eksempel skal kontrollere for hele strukturen af programmet. Desuden er der i forvejen vundet temmelig meget ved at specialisere programmet da hele udtrykket som nævnt er statisk og derfor bliver specialiseret væk så selv om tabet i værste fald er på 400% ved at bruge Pestos vogter skal man stadig tage i betragtning at programmet kører 4 gange så hurtigt med den specialiserede kode og vogteren end det oprindelige program gjorde. Og det har været meget lettere at specialisere programmet ved at benytte Pesto end ved at skrive JSpec-konteksten i hånden og selv sørge for at det specialiserede kode bliver kaldt.

5.5 Konfigurationsfil

Ud over *_JSpec_Context.java* skal JSpec bruge en konfigurationsfil med efternavnet *jspec*, en såkaldt *modulfil*. Filen skal indeholde et antal parametre som JSpec skal bruge under specialiseringen. Konfigurationsfilen kan genereres direkte fra specialiseringsklassefilen, og der er ikke brug for yderlig input fra brugeren. Et eksempel på en modulfil til AnyExpression kan ses i figur 5.31.

Den første linje i modulfilen er blot navnet på modulfilen hvori navnet på den fil specialiseringsklasserne ligger i indgår. Den næste linje er signaturen på indgangspunktet som også kan læses direkte i specialiseringsklasserne. Den tredje parameter, *entry_point_signature*, giver bindingstiderne på den specialiseringsklasse som indgangspunktet ligger i, det vil sige klassens **this**, samt bindingstiderne på indgangspunktets parametre i den rækkefølge de står i signaturen. Bindingstiden på **this** vil for alle praktiske formål altid være statisk (S) da det altid er kendt hvilken type rodklassen har. Hvis kun typen er kendt og indholdet af **this** er dynamisk, er **this** delvis statisk hvilket stadig er statisk. Generelt kan man sige at hvis en parameter betegnes som statisk (S), vil den forsvinde i den specialiserede kode da værdien af den i hele metodekaldet er kendt og kald til variablens værdi skiftet ud med den kendte værdi. Denne parameter vil altså slet ikke være der i den specialiserede kode hvorimod dynamiske parametre (D) stadig vil stå i signaturen til den specialiserede metode.

Når parametre (og lokale variabler) bliver specialiseret væk, er det noget der forgår i JSpec, så i sidste ende er det JSpec der afgør om en parameter eller variabel bliver specialiseret væk. Det er derfor JSpecs opførsel jeg skal efterligne når jeg skal afgøre hvorvidt parametre skal være statiske eller dynamiske.

Det er let at afgøre om en primitiv type er statisk eller dynamisk. Hvis værdien og en eventuel reference til værdien er kendt, er variablen statisk og ellers ikke. Anderledes svært er det med objekter – JSpec bruger jo en hel bindingstidsanalyse på at finde ud af det. Jeg har derfor lavet en approksimation der siger at parametre af en objekttype hvilket også vil sige arrays der indholder objekttyper altid er bliver bibeholdte. Hvis længden og indholdet af et array af en primitiv type er kendte, er hele arrayet kendt. Denne approksimation er rimelig god da det faktisk er sjældent at objekter er bliver specialiseret bort (i hvert fald i de eksempler jeg har kigget på). Men det sker, og det skal Pesto selvfølgelig kunne håndtere. For eksempel sker det i eksemplet med beregning af aritmetiske udtryk i specialiseringen AnyExpression hvor hele udtrykket ender med at være statisk. Det kan programmøren angive nederst i specialiseringsklassefilen med linjen:

entrypoint overwrite`: (S,D)` (27)

som angiver hvordan programmøren forventer bindingstiderne på parametrene *mindst* er. Parenteserne skal indeholde det samme antal parametre som metoden. I ovenstående eksempel forventer programmøren at den første parameter, `Exp` *e*, mindst er statisk, det vil sige at den bliver specialiseret væk. Den anden parameter, arrayet `float[]` *env*, angiver han til at være dynamisk, men det betyder blot at den mindst er dynamisk og muligvis statisk. Som det ses i figur 5.31, afgør Pesto at anden parameter er statisk.

Derefter angives navnet på den specialiserede metode som konstrueres fra navnet på metoden (_<*metodenavn*>_spec).

Den næste parameter i modulfilen er *main*-klasserne hvilket vil sige de klasser som JSpec skal bruge i specialiseringen, altså de klasser der bliver brugt af den del af programmet der skal specialiseres. Hvis det drejer sig om flere klasser end dem der er nævnt i specialiseringsklasserne, skal programmøren angive klasserne i bunden af specialiseringsklassefilen som *extra classes* som set i roboteksemplet:

extra classes`: {Event, SpeedEvent, TemperatureEvent}` (28)

De sidste tre parametre i modulfilen er nogle værdier JSpec bruger under specialiseringen og som ikke afhænger af specialiseringsklasserne. Der kan selvfølgelig forekomme specialiseringer hvor disse værdier (eller rettere nogle af dem) skal ændres, men de er netop valgt så det sjældent forekommer.

Pesto genererer desuden en anden konfigurationsfil som AspectJ skal bruge til at flette kode ind i klassefilerne. Filen er en liste over de javaklasser AspectJ skal bruge, plus aspektet, det vil sige filen med vogteren fra Pesto og den specialiserede metode som JSpec har genereret som et aspekt.

5.6 Implementation af Pesto-oversætteren

Til at implementere Pesto-oversætteren har jeg brugt en såkaldt *compiler-compiler* som ud fra en grammatik genererer klasser og metoder til at parse et program skrevet efter den givne grammatik og gennemløbe en opparset træstruktur. Jeg har brugt SableCC 2.16.2 [Gagnon98] som genererer et objektorienteret framework i Java hvor det er let til tilføje klasser til at generere oversættelsen. SableCC genererer et abstrakt syntakstræ (AST) og giver en klasse til at gennemløbe træet dybde-først ved at bruge Visitor-designmønstret.

Pesto-oversætteren gennemløber syntakstræet syv gange i følgende faser:

Weaver Grammatikken for de programmer oversætteren Pesto kan parse, er „større“ end den faktiske grammatik for Pesto. For eksempel kan man i følge grammatikken have mere end en specialiseringsklasse med indgangspunkt. Weaveren

kontrollerer om specialiseringsklasserne lever op til grammatikken, og giver en passende fejl hvis ikke.

CollectionVisitor Alle oplysninger bliver indsat i en symboltabel til opslag fra andre visitorer. Hver specialiseringsklasse har sit eget objekt af klassen SpecClass med oplysninger om specialiseringsklassen. En specialiseringsklasse har ud over navn og rodklasse et antal prædikater og eventuelt et indgangspunkt. Et prædikat og en metode (indgangspunktet) har deres egne klasser, og alle variabler i strukturen kan kun kaldes gennem metoder defineret i SpecClass. Denne struktur har gjort det lettere at rette oversætteren til og lave eventuelle ændringer i koden. Det er kun den information der kan læses direkte i specialiseringsklasserne der opsamles i CollectionVisitor. Næste fase sørger for resten.

CheckTypesVisitor Typerne på variablerne i en specialiseringsklasse kontrolleres for om de er konsistente med de faktiske typer i rodklasserne, og det kontrolleres at rodklasserne er definerede. Desuden opsamler CheckTypesVisitor typerne på prædikater og tilføjer dem i symboltabellen.

WriteTemplate Værdifilskabelonen som senere skal rettes af brugeren skrives til en fil, *template<specclass-fil-navn>.values*

SetAnalysisContext Metoden setAnalysisContext hvori analysekonteksten sættes skrives i klassen _JSpec_Context

SetSpecializationContext Metoden setSpecializationContext som sørger for specialisering og generering af vogtere tilføjes til _JSpec_Context.

WriteConfiguration Modulfilen *pesto_<specclass_name>_mdl.jspec* og klasselisten *pesto_<specclass_name>_argfile.lst* til brug af henholdsvis JSpec og AspectJ skrives.

Jeg har forsøgt at bruge så meget af informationen direkte fra syntakstræet i stedet for at slå op i symboltabellen. Derved har jeg fået kode der er lettere at debugge og ændre.

Når man har skrevet så stort et program som en oversætter, får man typisk til sidst i processen et utrolig godt overblik og vil derfor have gode forudsætninger for at skrive oversætteren om fra bunden til at være mere elegant og overskuelig da mange designvalg taget tidligt i processen har betydning for resten af designet – dertil er jeg også nået og ville med den indsigt jeg har nu kunne skrive en bedre oversætter dermed gøre vedligeholdelsen af den lettere.

Hvordan bruges Pesto

Det burde være klart at den optimale måde at benytte programspecialisering på er at skrive programmet med henblik på specialiseringen. Når programmet er skrevet og

oversat, skrives de tilhørende specialiseringsklasser i en fil med et sigende navn og efternavn efternavnet „*.sc*“. Derefter kaldes Pesto til analysen af programmet med linjen:

horse01:~/Robot% pesto -analyze MyRobot.sc (29)

Kommandoen „pesto“ er et script der kalder Pesto i dette tilfælde med specialiseringsklassefilen *MyRobot.sc*, og derefter kalder JSpecs analysefase. Når analysen er færdig – og det kan godt tage lang tid, ændres skabelonfilen af brugeren til at indeholde de faktiske værdier, og filen gemmes for eksempel som *MyRobot.values*. Så er programmet parat til at blive specialiseret:

horse01:~/Robot% pesto -specialize MyRobot.sc -values MyRobot.values (30)

Efter specialiseringen ligger det specialiserede kode og vogterene i en fil *MyRobot.java*, og filen flettes sammen med den oprindelige kode med kommandoen:

horse01:~/Robot% pesto -weave MyRobot.java (31)

Og derefter kan programmet køres som før det blev specialiseret.

Hvis man ønsker det og alle prædikaterne i specialiseringsklasserne er eksplicit statiske kan disse tre faser køres på en gang ved blot at kalde Pesto med specialiseringsklasserne.

5.7 Sammenfatning

I dette kapitel har jeg gennemgået nogle af de væsentlige emner ved selve konstruktionen af Pesto, nemlig den måde hvorpå Pesto arbejder sammen med den partielle evaluator, JSpec. Jeg beskrev hvordan jeg genererer specialiseringskonteksten til JSpec og hvordan jeg i specialiseringskonteksten genererer den vogterkode der sørger for at det specialiserede kode bliver kaldt på det rigtige tidpunkt. Ud fra eksempler har jeg redegjort for at det er vigtigt at overveje valg af indgangspunkt – eventuelt ændre den kode man ønsker at specialisere så indgangspunktet ligger fri af den kritiske kode. Jeg viste også at det overhead som vogteren skaber i det endelige program er acceptabelt i forhold til hvad man vinder ved ikke selv at skulle sørge for at kalde det specialiserede kode. Desuden har jeg ved hjælp af semantikken givet et billede på hvordan oversætteren er bygget op. Det er i dette kapitel man kan få en fornemmelse for hvad brugeren skånes for ved at skrive specialiseringsklasser i Pesto fremfor at skrive kontekst og konfigurationsfil i hånden og desuden selv sørge for at kalde koden.

6 Afslutning

Dette er begyndelsen af slutningen – det sidste kapitel i mit speciale hvor alle løse ender skal samles. Jeg vil begynde med at gennemgå nogle af de projekter som behandler nogle af de samme problemstillinger som jeg har gjort i dette speciale. Derefter vil jeg gennemgå forskellige muligheder for udvidelser af Pesto. Til sidst vil jeg diskutere perspektiverne for Pesto og konkludere på dette speciale.

6.1 Relaterede forskningsområder

I dette kapitel vil jeg redegøre for et udsnit af de områder der har betydning for mit speciale og for udbredelsen af partiel evaluering.

Partiel evaluering

En stor del af dette speciale bygger på partiel evaluering og som jeg nævnte indledningsvist fortrinsvist *offline* partiel evaluering. I dette kapitel vil jeg give en kort beskrivelse af *online* partiel evaluering.

Offline partiel evaluering har to faser; en analyse- og en specialiseringsfase. Online partiel evaluering har derimod kun en fase – bindingstidsanalysen og specialiseringen sker så at sige samtidig. Begge former har fordele og ulemper; online specialisering er den mest præcise da alle oplysninger om programmet er til rådighed til specialisatoren, men tager længere tid om specialiseringen da de statiske dele af programmet fortolkes under specialiseringen. Offline specialisering er derimod den mest effektive af de to, da det under analysen afgøres hvilken kode der kan evalueres på oversættelsestidspunktet og hvilken der må vente til udførselstidspunktet. Denne afgørelse er en approksimation og derfor ikke så nøjagtig som online partiel evaluering [Consel93, Jones93]. JSpec er naturligvis en offline partiel evaluator.

Deklarativ specialisering

Der er andre tiltag inden for deklarativ sprog til specialisering, og jeg har valgt nogle stykker ud der er relevante for Pesto. Først og fremmest er der Volanschis *Specializa-*

tion Classes som er de oprindelige specialiseringsklasser som Pesto bygger på. I det følgende vil jeg gennemgå de vigtigste tiltag.

Specialization Classes

Pesto og Specialization Classes minder naturligvis meget om hinanden i opbygningen, da de begge har en objektorienteret tilgang, og eftersom jeg i designet af Pesto har bygget videre på Specialization Classes er sammenfaldet ikke overraskende. Men der er dog nogle forskelle ud over de ændringer i syntaksen som jeg har beskrevet i specialet (se for eksempel kapitel 4.3).

I Specialization Classes er det ikke muligt at lave typeerklæringer, det vil sige at specialisere for at en instansvariabel har en bestemt type og det er en af de store forskelle på Pesto og Specialization Classes da dette har betydning for at kunne specialisere interaktionen mellem objekter som må siges at være afgørende i objektorienterede programmer, og et af de steder specialiseringen betyder meget som jeg har redegjort for tidligere. Desuden har den manglende præcision med hensyn til typeerklæringer betydning for konstruktionen af vogtere og nedarvning af specialiseringsklasser.

Specialization Classes har en konstruktion som jeg har udeladt i Pesto, nemlig at give mulighed for at erklære om specialiseringen skal ske på *runtime* eller *compile time*. Det kan være en fordel at lade specialiseringen ske på runtime hvis man ikke kender de faktiske værdier før. JSpec understøtter dog ikke specialisering på runtime og derfor er det udeladt af Pesto.

Implementationen af Specialization Classes er lavet før JSpec var færdigkonstrueret og derfor oversætter Specialization Classes ikke til specialiseringskontekst. Specialization Classes omfatter som Pesto også vogtere, men af en lidt anden type end dem jeg har beskrevet, da vogteren er på den enkelte instansvariabel som en sandhedsværdi om hvorvidt prædikatet på denne instansvariabel er opfyldt eller ej, og at denne vogter kun skal opdateres når instansvariablens værdi ændres. Vogterene giver dermed kun mening for private instansvariabler, da det er nødvendigt at vide hvor i programmet der tilordnes en værdi til variablen og det kan man kun vide med private instansvariabler med mindre man inkluderer hele Javas klasserhierarki. Restriktionerne om private instansvariabler giver kraftige begrænsninger i forhold til håndtering af vogtere på arrays, og dette må også siges at være en ulempe da arrays ofte også har afgørende betydning i specialiseringer som det blandt andet sås i roboteksemplet.

Vogterne bliver på denne måde ikke kontrolleret hver gang en metode kaldes hvilket i tilfælde hvor instansvariablerne sjældent skifter kan være en fordel. Specielt hvis der er mange prædikater i en specialisering som skal kontrolleres hver gang indgangspunktet kaldes [Volanschi98a]. Jeg har også overvejet denne metode som jeg vil redegøre for i

kapitlet om perspektiverne for Pesto.

Pesto er en udvidelse af Specialization Classes og har derfor meget mere funktionalitet. Blandt andet har det vist sig at invarianter på typer er meget brugbart [Schultz99, Lawall99, Schultz00, Schultz02]. Desuden er vogterne Pesto mere generelle end dem i Specialization Classes hvor vogtere kun kan sættes på private instansvariabler med en primitiv type.

Specialiseringsmoduler

Specialiseringsmoduler er en metode til at erklære en specialiseringskontekst til specialisering af C-programmer [Meur02b]. I stedet for specialiseringsklasser bygger sproget på specialiseringsscenarier hvor man for en givet funktion i sit program kan angive bindingstiderne for metodens parametre. I specialiseringsmodulerne kan man angive bindingstider for parametrene i flere funktioner, men det er ikke muligt at lave prædikater på lokale variabler i specialiseringsmodulerne. Under analysen kontrolleres det at bindingstiderne bliver det samme under hele programudsnittet, men specialiseringsmoduler har ikke vogtere så programmøren skal selv sørge for at kalde de specialiserede funktioner. Til specificeringen af de faktiske specialiseringsværdier er der udviklet et grafisk værktøj [Meur02a].

Specialiseringsmoduler er konstrueret til specialisering af C-programmer, og allerede der er der en væsenlig forskel til Pesto. Syntaksen til modulerne virker kompliceret og uoverskuelig i forhold til specialiseringsklasserne. Det skal dog siges at hvis man først har lært syntaksen og har en god viden om partiel evaluering er specialiseringsmoduler et godt deklarativt sprog til partiel evaluering – det grafiske værktøj er en god hjælp til brugeren.

Prædikatklasser

En anden tilgang til at erklære prædikater på klasser er prædikatklasser (*predicate classes*) [Chambers93] hvor det er muligt at erklære tilstandsinformation om instansvariabler. En prædikatklasse repræsenterer delmængden af instanser af dens subklasse der opfylder prædikaterne, og klassen arver metoder og instansvariabler fra prædikatklassen. Prædikatklasser er en generalisering af multidispatching, og kan, modsat Pesto, tilføje ny funktionalitet til sine superklasser. Man kunne godt bruge prædikatklasser til at implementere specialiseringsklasser, da specialiseringsklasser er en slags delmængde af prædikatklasser som kun kan bruges til specialiseringen. Vogterne ville da allerede være implementeret via multidispatching, men problemet ville være at skabe den specialiserede kode som skal angives i prædikatklassen. Prædikatklasserne kan kun overtage Pestos funktion i forhold til specialisering. Det vil stadig være nødvendigt at anvende partiel evaluering til den faktiske specialisering.

Specialiseringsmønstre

Som beskrevet tidligere giver designmønstre mange fordele, for eksempel generalitet, genbrug, uafhængighed og så videre. Men de skaber også et stort overhead da de typisk indeholder en masse små objekter. Jeg beskrev også hvordan partiel evaluering kan bruges til at eliminere noget af dette overhead, og faktisk er dette område formaliseret i *specialiseringsmønstre* [Schultz00] som giver et mønster for hvordan man ved hjælp af et Pesto-lignende sprog kan specialisere et antal designmønstre og på den måde få glæde af fordelene ved designmønstre uden nødvendigvis at skabe et stort overhead.

6.2 Perspektiver

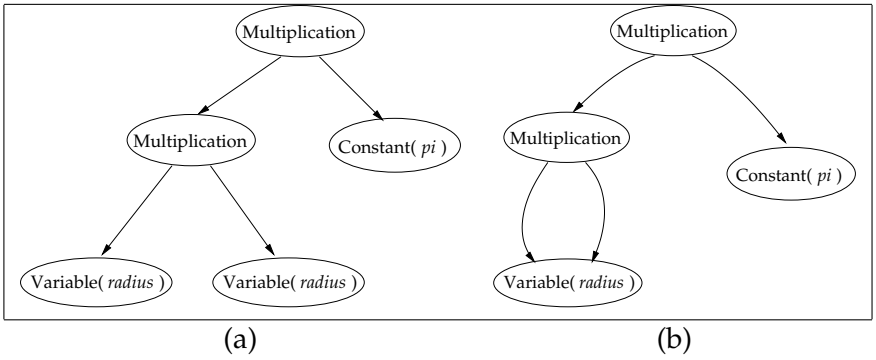
Under udvikligen af Pesto er der dukket nogle interessante problemstillinger op som jeg ikke har behandlet i specialet, enten fordi det ikke har være direkte relevant for projektet eller fordi det har været for stor en mundfuld at implementere. Jeg vil i dette kapitel beskrive nogle af disse områder.

Subtyper

I det oplæg jeg oprindelig lavede til Pesto var det muligt at erklære at en variabel havde en bestemt type *eller* en subtype til typen med notationen:

$$v{:<}\ T; \tag{32}$$

Denne syntaks er ikke kendt fra Java, men bruges i andre programmeringssprog til at udtrykke subtypesammenhænge (for eksempel Beta). Det viste sig dog ikke at være helt trivielt at oversætte til `_JSpec_Context`. I analysekonteksten er der to måder at specificere en instansvariabels aliasrelationer. Den første er den der bruges hvis en typeerklæring er eksplicit statisk hvor instansen af variablen tilordnes en ny instans af den på gældende type ved at bruge **new**. Men dette bruges til at specificere at instansvariablen har denne type og præcis denne type og ikke en subtype. Den anden måde er den der bliver brugt når typeerklæringen er dynamisk, det vil sige når der ikke er noget prædikat på instansvariablen. Her bruges metoden `get_object` i klassen `DynamicValue`. Denne metode returnerer et ny instans af typen `Object` eller en hvilken som helst subklasse af `Object` som forekommer i programmet. Man kunne forestille sig en tilsvarende metode i `StaticValue`-klassen med dette ville stadig være for upræcist da det naturligvis kun er en delmængde af `Object` der har interesse. Løsningen er givet i JSpecs *magic context interface* som er konstrueret netop til at løse nogle af de problemer der opstår i analysekonteksten. Interfacet giver for en given klasse mulighed for at give en instans af et objekt *eller* en subklasse af objektet, statisk eller dynamisk, og det er netop det der er brug for. Interfacet giver desuden også mulighed for at give en instans af klassen selv om klassen ikke har en *default constructor* hvilket løser en af restriktionerne på det program der



Figur 6.1: Strukturer af udtryk der beregner arealet af en cirkel.

skal specialiseres, nemlig at alle klasser skal have en default constructor. Det er trivielt at ændre Pesto så subtyper bliver understøttet, men jeg har udeladt det da det magiske interface endnu ikke er helt pålideligt.

Samme objekt

I kapitel 4.8 beskrev jeg en specialisering af beregning af aritmetiske udtryk med hensyn til arealet på en cirkel. Specialiseringen blev ikke udløst af en hvilken som helst beregning af arealet af en cirkel, men af en der havde den struktur der var angivet i specialiseringsklasserne (figur 6.1), bortset fra at Pesto ikke kontrollerer om to instansvariabler refererer til det samme objekt eller ej, og at specialiseringen derfor også bliver udløst af strukturen i figur 6.1 (b). Sagt på en anden måde; hvis programmet er en *DAG* (*directed acyclic graph*) (som figur 6.1 (b)) kontrollerer Pesto for den udfoldede graf (figur 6.1 (a)). Hver knude i DAG'en er et objekt og en kant er en reference fra en instansvariabel i et objekt til et andet objekt.

Pesto vogter altså kun for at indholdet af objekterne er det rigtige, men det ville være en relevant udvidelse af Pesto at have mulighed for at specialisere med hensyn til at to instansvariabler indeholder det samme objekt, eller netop ikke indeholder det samme objekt. JSpec specialiserer for det den bliver bedt om, det vil sige det der bliver beskrevet i specialiseringsklassefilen/værdifilen. I de fleste tilfælde er det ikke noget problem at JSpec har specialiseret for at to objekter er det samme og de rent faktisk blot er ens. Men hvis der et sted i programmet er en sammenligning mellem referencerne til objekterne giver det selvfølgelig problemer, da der muligvis er specialiseret for denne sammenligning. Så ændringen vil medføre at vogteren bliver semantisk korrekt i forhold til specialiseringen. En anden fordel ved at kunne specialisere for at to referencer peger på det samme objekt er at vogteren kun behøver blive kontrolleret en gang, da der for de efterfølgende referencer blot skal kontrolleres at referencen er den samme.

```
AnyExpression {
    calc(e) : SpecMult;
    calc(env) : float[1];
}

SpecMult {
    left : SpecMult#1;
    right : SpecConst;
}

SpecMult#1 {
    left : SpecVar;
    right : SpecVar#1;
}

SpecConst {
    value = 3.1416F;
}

SpecVar {
    identifier = 0;
}

SpecVar#1 {
    identifier = 0;
}
```

(a)

```
AnyExpression {
    calc(e) : SpecMult;
    calc(env) : float[1];
}

SpecMult {
    left : SpecMult#1;
    right : SpecConst;
}

SpecMult#1 {
    left : SpecVar;
    right : SpecVar;
}

SpecConst {
    value = 3.1416F;
}

SpecVar {
    identifier = 0;
}
```

(b)

Figur 6.2: To forskellige version af de faktiske værdier til beregningen af arealet af en cirkel.

```
private static Variable[] sameObject_SpecVar;

private boolean Calculate._guard_AnyExpression(Exp e, float[] env){
  sameObject_SpecVar = new Variable[1];
  if (!(e.getClass() == Multiplication.class)) return false;
  if (!(((Multiplication)e)._guard_SpecMult0())) return false;
  if (!(env.getClass() == float[].class)) return false;
  return true;
}

private boolean Multiplication._guard_SpecMult0(){
  if (!(this.left.getClass() == Variable.class)) return false;
  if (sameObject_SpecVar[0] == null) {
    if (!(((Variable)this.left)._guard_SpecVar0())) return false;
    sameObject_SpecVar[0] = (Variable)this.left;
  } else {
    if (sameObject_SpecVar[0] != (Variable)this.left) return false;
  }
  if (!(this.right.getClass() == Variable.class)) return false;
  if (sameObject_SpecVar[0] == null) {
    if (!(((Variable)this.right)._guard_SpecVar0())) return false;
    sameObject_SpecVar[0] = (Variable)this.right;
  } else {
    if (sameObject_SpecVar[0] != (Variable)this.right) return false;
  }
  return true;
}

private boolean Variable._guard_SpecVar0(){
  if (!(this.identifier == 0)) return false;
  return true;
}
```

Figur 6.3: Udsnit af en vogter for der kontrollere at de to variabelobjekter er det samme.

Syntaksen for sådan en udvidelse ligger lige for. Hvis to variabler refererer til den samme specialiseringsklasse skal instansvariablerne indeholde det samme objekt og ellers ikke. Figur 6.2 (a) viser værdifilen til et program med to forskellige variabelobjekter og figur 6.2 (b) viser værdifilen for specialiseringen af at de to instansvariabler peger på det samme objekt.

Figur 6.3 viser vogteren for værdifilen i figur 6.2 (b) hvor de to variabelobjekter skal specialiseres for at være det samme. Ideen er at opretholde et array (*sameObject_SpecVar*) af samme type som specialiseringsklassens rodklasse (Variable), og med samme længde som antallet af forskellige versioner af specialiseringsklassen (i dette til SpecVar) – dette antal kan findes ved at søge værdifilen igennem for antal forekomster af navnet på spe-

cialiseringsklassen. Hver indgang i arrayet repræsenterer en specialiseringsklasse og initialiseres til **null**. Når en reference til en specialiseringsklasse skal kontrolleres, slås op i arrayet for at se om der allerede er et objekt. Hvis ikke, kontrolleres det om vogteren på objektet er opfyldt og hvis den er, puttes objektet ind i arrayet. Næste gang et objekt med samme prædikat skal kontrolleres slås op i arrayet og hvis der er et objekt, sammenlignes referencen på de to og hvis den peger på samme objekt er vogterens betingelse opfyldt. Hvis ikke den peger på samme objekt, har programmet en struktur som den i figur 6.1 (a) og det specialiserede kode skal ikke udføres.

Der er dog et problem med denne fremgangsmåde; hvis der specialiseres for at to instansvariabler peger på to forskellige objekter af samme type (som i figur 6.2 (b)) og der i det program der køres peges på samme objekt (som i figur 6.1 (b)), køres den specialiserede kode alligevel da vogteren blot putter det samme objekt ind i hver sin indgang i arrayet og ikke kontrollerer om det rent faktisk er det samme objekt. At kontrollere for at hver objekt ikke er det samme som en af de andre ved at sammenligne, giver en udførselstid på n^2 hvor n er antallet af objekter så denne løsning dur ikke.

Jeg kan med andre ord med denne metode kun kontrollere at en graf (parsertræ) er en udfoldelse af den DAG jeg har og at de DAG-knuder[1] jeg ved skal være der rent faktisk er der. Jeg kan ikke kontrollere at der er for mange DAG-knuder. Så jeg vil gerne kunne kontrollere at der kun er de DAG-knuder der skal være i programmet. Da jeg ved at grafen er næsten korrekt, bortset fra DAG-knuderne, kan jeg opfylde den sidste betingelse ved at tælle antallet af samme objekt af en bestemt klasse. Der kræver naturligvis at jeg ved på forhånd hvor mange objekter der skal være af en givet klasse, men det kan undersøges ved at parse specialiseringsklassefilen og værdifilen.

Derfor skal vogteren også holde øje med at der er det rigtige antal objekter der opfylder hver specialiseringsklasse – en sådan vogter der kontrollerer for at programmet har to variabelobjekter, ses i figur 6.4[2] (af pladshensyn har jeg erstattet noget af koden med „…“ – koden er ækvivalent med den ovenover, blot med første indgang i arrayet i stedet for nulte). Arrayet *sameObject_SpecVar* har her længden 2 da der er to versioner af specialiseringsklassen SpecVar, og der skal være netop en repræsentation af hver objekt. Rent praktisk løses problemet med at tælle objekter ved at der i hver klasse (her Variable) ved hjælp af AspectJ tilføjes en instansvariable „boolean *seenAll*“ som er „true“ hvis alle instanser af dette objekt er set af vogteren, og at der i vogteren tilføjes endnu to arrays af int af samme længde som *sameObject_SpecVar* per klasse; et der for hver version af specialiseringsklassen indeholder det antal objekter der skal være (*numberOf_SpecVar*) og et der indeholder det antal der indtil videre er set (*seenSoFar_SpecVar*). Når de to tal i

[1] en knude der gør grafen til en DAG, det vil sige en knude der har mere end en kant ind til sig.

[2] Faktisk burde de samme tjek foretages i vogteren for SpecMult0, men af pladshensyn har jeg udeladt dem her.

```
private boolean Multiplication._guard_SpecMult0(){
  if (!(this.left.getClass() == Multiplication.class)) return false;
  if (!(((Multiplication)this.left)._guard_SpecMult1())) return false;
  if (!(this.right.getClass() == Constant.class)) return false;
  if (!(((Constant)this.right)._guard_SpecConst0())) return false;
  return true;
}
private boolean Multiplication._guard_SpecMult1(){
  if (!(this.first.getClass() == Variable.class)) return false;
  Variable this_first = (Variable)this.first;
  if (numberOf_SpecVar[0] != seenSoFar_SpecVar[0]) {
    if (!this_first.seenAll) {
      if (seenSoFar_SpecVar[0] > 0) {
        if (!(sameObject_SpecVar[0] == this_first)) return false;
        seenSoFar_SpecVar[0]++;
        if (seenSoFar_SpecVar[0] == numberOf_SpecVar[0]) {
          this_first.seenAll = true;
        }
        sameObject_SpecVar[0] = this_first;
      } else {
        seenSoFar_SpecVar[0]++;
        if (seenSoFar_SpecVar[0] == numberOf_SpecVar[0]) {
          this_first.seenAll = true;
        }
        sameObject_SpecVar[0] = this_first;
      }
    } else {
      return false;
    }
  }
  if (!(this.first.getClass() == Variable.class)) return false;
  Variable this_second = (Variable)this.second;
  if (numberOf_SpecVar[1] != seenSoFar_SpecVar[1]) {
    if (!this_second.seenAll) {
      if (seenSoFar_SpecVar[1] > 0) {
        ...
      } else {
        ...
      }
    }
  }
  for (int i = 0; i < sameObject_SpecVar.length; i++) {
    if (!sameObject_SpecVar[i]) return false;
  }
  return true;
}
private boolean Variable._guard_SpecVar0(){
  if (!(this.identifier == 0)) return false;
  return true;
}
private boolean Variable._guard_SpecVar1(){
  if (!(this.identifier == 0)) return false;
  return true;
}
```

Figur 6.4: Udsnit af vogter der kontrollerer at de to objekter er forskellige.

en givet indgang er ens, sørger vogteren for at sætte *seenAll* til „true“ i objektet i den tilsvarende indgang i *sameObject*-arrayet. Inden vogteren returnerer, skal den kontrollere at alle *seenAll* er „true“ og hvis det er tilfældet, har programmet nøjagtig den struktur der specialiseres for.

Variabelvogtere

En vigtig del af Pesto er vogterne der sørger for at den rigtige kode bliver udført på det rigtige tidpunkt. Og det er naturligvis vigtigt at det at udføre vogteren ikke koster så meget at det ikke kan betale sig at specialisere.

Som beskrevet i kapitel 5.4 har jeg implementeret vogteren på indgangspunktet så prædikaterne bliver kontrolleret hver gang indgangspunktet kaldes. Ulempen ved denne metode er at der muligvis bliver udført uhensigtsmæssigt mange tjek hvis indgangspunktet kaldes ofte og muligvis unødvendig mange tjek hvis variablerne ikke ændrer sig. I værste fald kan vogterne skabe så stort et overhead at fordelen ved at specialisere forsvinder.

En anden måde at implementere vogterne på er at sætte en vogter på hver af de instansvariabler der er et prædikat på. Det er denne metode Volanschi foreslår. Jeg har kaldt denne metode for variabelvogter og den anden metode for metodevogter.

Ideen bag variabelvogtere er at hver objekt ved om det potentielt er i en tilstand der passer til en givet specialiseringsklasse, det vil sige om objektets instansvariabler har de rette typer og for primitive typer, de rigtige værdier. Inden indgangspunktet udføres skal det kontrolleres at hver instansvariabel der indgår i en specialiseringsklasse har den rigtige type og hvis det er tilfældet er vogteren opfyldt. Det vil i værste fald sige at der skal udføres et tjek af en boolean for hver prædikat i specialiseringsklasserne, men dette er stadig en meget billigere operation end de sammenligninger der foretages i metodevogterne. Rent praktisk skal der for hver prædikat i hver specialiseringsklasse ved hjælp af AspectJ introduceres en boolean i instansvariablens klasse. Hvis instansvariablen er i en potentiel tilstand, det vil sige at den har den rigtige værdi hvis den er af primitiv type og ellers at den har den rigtige type. Hvis prædikatet er en typeerklæring til en specialiseringsklasse er det stadig nok at kontrollere om instansvariablen har den rette type – det objekt der refereres til ved selv om det er i den rette (potentielle) tilstand. Hvis instansvariablen er i en potentiel tilstand sættes den relevante boolean til „true“. Eventuelt kan der for hver (forskellig) specialiseringsklasse der specialiserer en givet klasse introduceres en ekstra boolean som afspejler om alle klassens instansvariabler er i en potentiel tilstand eller ej. Derved tager det lidt længere tid hver gang en instansvariabel ændres, men kortere tid når indgangspunktet udføres hvilket jo er målet med denne type vogtere.

AspectJ har en metode **set** som givet en klasse og en instansvariabel i klassen giver et pointcut til når instansvariablen bliver sat. På samme måde som for metoder (og **call**) er der herefter mulig at skrive en stump kode der bliver udført før eller efter en værdi tilordnes instansvariablen. Så når en instansvariabel med et prædikat ændres, kontrollerer vogteren om instansvariablen opfylder prædikatet og hvis den gør, sættes den tilsvarende boolean. Hvis man vælger at have endnu en boolean per specialiseringsklasse som beskrevet ovenfor, skal alle prædikatvariabler boolean kontrolleres til man finder en der er „false“, og hvis ikke skal den overordnede boolean sættes til true.

Fordelen ved denne form for variabelvogtere frem for dem som Volanschi foreslår [Volanschi97], udover at mit forslag også understøtter vogtere på typer, er at de ikke er afhængige af at instansvariablerne er private hvilket jo ikke er muligt når man specialiserer med JSpec der kræver at instansvariabler er public.

Jeg har ikke haft mulighed for at eksperimentere med variabelvogtere, men mit umiddelbare indtryk er at det ville fungere og være mere effektivt end metodevogtere.

Værktøj til ændring af filer

Jeg har beskrevet at det er nødvendigt for brugeren at ændre i den skabelonfil som Pesto skriver så den indeholder de faktiske værdier. Brugeren kan ændre i tekstfilen med en teksteditor (for eksempel emacs), men det er ikke særlig optimalt da det er afgørende når værdierne skal læses at det er de rigtige ting der bliver ændret. En måde at sikre det på er ved at konstruere et grafisk værktøj der hjælper brugeren.

Jeg forestiller mig at et sådan værktøj skal bestå af to niveauer. Det ene niveau skal give brugeren mulighed for at se skabelonfilen som den ser ud med klasserne, men det andet niveau skal give et grafisk billede af sammenhængen mellem specialiseringsklasserne, og det skal være muligt for brugeren at ændre strukturen både på det grafiske niveau og tekstniveauet. Det grafiske niveau skal give en grafstruktur af specialiseringen som svarer til dem jeg har brugt til at illustrere strukturen af beregningen af aritmetiske udtryk i figur 6.1. Et sådan værktøj vil gøre specialiseringen endnu mere tilgængelig for ikke-eksperter hvilket jo er et af mine erklærede mål med dette speciale.

Programspecialisering af pladshensyn

I indledningen brugte jeg små computere uden ret meget hukommelse som motivation for at bruge programspecialisering selv om Pesto i sin nuværende form blot indsætter det specialiserede kode i det oprindelige program og dermed gør det modsatte, nemlig at forøge programmets størrelse.

Jeg har valgt ikke at beskæftige mig med reduktion af programstørrelsen i dette spe-

ciale, ikke fordi det ikke er et interessant område, tværtimod, men fordi det er en delmængde af Pestos virkeområde.

Vogtere er stadig en vigtig del af specialiseringen da de skal sørge for at programmet ikke bliver misbrugt ved at blive brugt med andre værdier end de der er specialiseret med hensyn til. Men i stedet for at kalde den oprindelige metode hvis vogterne ikke er opfyldt, skal programmet give en fejl.

Rent praktisk sættes den specialiserede kode og den ændrede vogter ind med AspectJ nøjagtig som ellers. Derefter fjernes det kode der ikke længere er brug for, det vil sige den oprindelige kode, ved at bruge et værktøj. Et eksempel på et sådan værtøj er Jax [Tip99] der givet en mængde af klasser, fjerner unødvendige metoder, klasser og instansvariabler og ved forskellige metoder reducerer koden.

Mangler i Pesto

Udover de ting jeg har nævnt i de forgående kapitler som mulige forbedringer af Pesto, er der endnu en ting som ville gøre specialiseringen lettere for programmøren: gode fejlbeskeder fra Pesto-oversætteren. Oversætteren giver fejlbeskeder og nogle af dem giver også god information om hvad der er galt med programmet eller specialiseringsklasserne, men det er et område der har været nedprioriteret. Gode fejlbeskeder giver programmøren en mulighed for umiddelbart at finde sin fejl og rette den uden at behøve at debugge eller sætte sig ind i hvad oversætteren gør hvornår. Som oversætteren fungerer nu kan man bede den om at udskrive hvor i oversættelsen den befinder sig, det vil sige i hvilken fase og i hvilken knude, og jeg har naturligvis brugt debuginformation til at skrive oversætteren.

Men en god fejlbesked skal ikke kun benævne *hvor* fejlen er men og *hvad* fejlen er. Det gør Pesto desværre ikke i særlig mange tilfælde.

6.3 Konklusion

Partiel evaluering er svært at bruge. Det er nødvendigt med et indgående kendskab til teknikken for at være i stand til at skrive en specialiseringskontekst hvilket gør at programmører ikke tager metoden i betragtning når de skal skrive et program.

Jeg har med dette speciale introduceret Pesto som er et deklarativt sprog til specialisering af objektorienterede programmer. Målet er at gøre programspecialisering tilgængelig for ikke-eksperter. Hvor godt partiel evaluering fungerer har ikke været et af områderne for specialet, det lader jeg eksperterne om, jeg benytter det blot. Og hvis partiel evaluering via deklarative sprog bliver mere udbredt vil indsatsen på området sandsynligvis også øges til glæde for alle.

Jeg har konstrueret sproget ud fra det objektorienterede paradigme for at understøtte den forståelse programmøren har af sit program ved at lade en specialiseringskontekst være opbygget af specialiseringsklasser der som objektklasser indholder erklæringer om klassens instansvariabler.

Programmørens eneste bekymring er at specificere hvilke værdier der skal specialiseres for, så hans program opnår den bedst mulige specialisering. Det kan naturligvis ikke automatiseres da det kun er programmøren og ikke oversætteren der ved i hvilke omgivelser programmet skal køre. Programmøren kan nøjes med at have et minimum af kendskab til programspecialisering for at kunne strukturere sit program og sin specialisering på den mest hensigtsfulde måde.

Jeg er naturligvis ikke den eneste der har set problemet i de komplicerede specialiseringskontekster til programspecialisering, men udover brugervenligheden i Pesto, er et af mine bidrag til området *vogterene* hvis opgave med at dirigere koden tidligere har været endnu et område programmøren skulle bekymre sig om ved programspecialisering. Specialization Classes har også en slags vogtere men meget begrænsede da der ikke er præcise typeerklæringer og vogterne på instansvariabler er begrænset af at instansvariablerne skal være private. Desuden har jeg integreret Pesto fuldstændig med en partiel evaluator hvilket jeg kun har set i specialiseringsmodulerne som til gengæld ikke implementerer vogtere. Ydermere er jeg kommet med et forslag til hvordan vogtere kan forbedres så de for nogle programkontekster generere mindre overhead end dem jeg har implementeret.

Hvis man ønsker at specialisere programmer med flere forskellige specialiseringskontekster er det utrolig simpelt at håndtere i Pesto der for hver specialisering genererer et privat, selvindeholdt modul med det specialiserede kode og tilhørende vogtere. Programmøren skal blot vælge hvilke af de specialiserede scenarier han ønsker og så sørger AspectJ for at holde versionerne adskilt.

Selve implementationen af Pesto (med forbehold for små uopdagede fejl) dækker alle de funktioner jeg har beskrevet i dette speciale og ved at have så udtryksfuldt et sprog har jeg haft rig mulighed for at teste Pesto på rigtige programmer hvilket har givet mig en god fornemmelse for sprogets problemområde med svagheder og styrker.

Desuden har brugervenligheden i sproget været et meget stort interesseområde for mig og jeg synes det er lykkedes at gøre sproget let at bruge og let at forstå.

Alt i alt er Pesto et helt værktøj der rammer et reelt behov og som er let at gå til. De problemer der eventuelt er med værktøjet stammer fra problemer med den bagvedliggende partielle evaluator, JSpec, som på nogle områder er tung at håndtere og til tider skal behandles på den helt korrekte måde.

Jeg har gennem specialet givet læseren et indblik i hvad man kan vinde ved at bruge Pesto, både i forhold til det der kan måles, nemlig udførselstiderne men i høj grad også i forhold til den tid det tager for programmøren at benytte specialisering ved at bruge *deklarativ specialisering af objektorienterede sprog*.

Helle Markmann
helle@markmann.dk

A Designmønstret „Observer“

Observer-designmønstret er beskrevet af Gamma mfl., og denne beskrivelse tager udgangspunkt i deres præsentation [Gamma95, side 293-303].

Formål

At definere en en-til-mange relation mellem objekter så når et objekt ændrer tilstand vil alle de objekter der afhænger af objektet automatisk blive underrettede og opdaterede.

Problem

Observer-mønstret kan bruges når:

- En abstraktion har to forskellige dele hvor den ene afhænger af den anden. Hvis disse to dele indkapsles i separate objekter, kan de ændres og genbruges uafhængig af hinanden.
- En ændring i et objekt kræver at et ukendt antal objekter skal ændres.
- Et objekt skal være i stand til at underrette andre objekter uden at gøre sig antagelser om hvilke type objekter det drejer sig om.

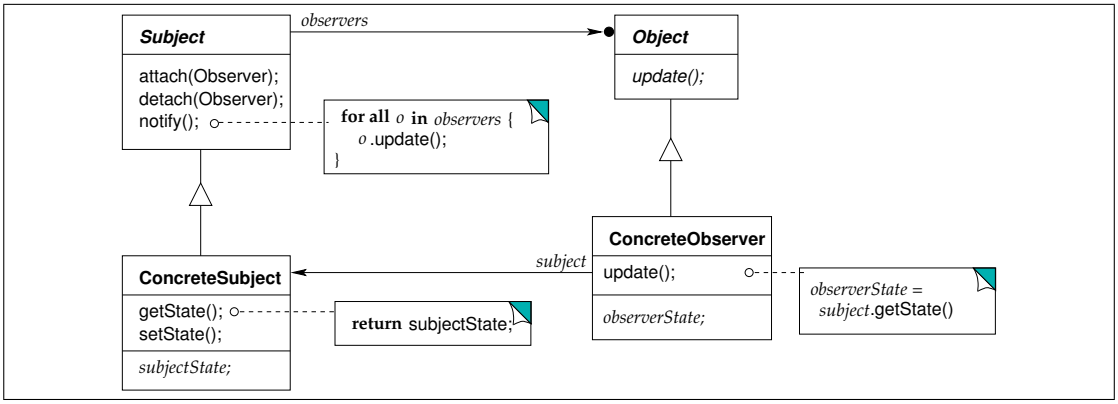
Struktur

Figur A.1 viser strukturen af observermønstret [Gamma95]. Beskrivelsen af de enkelte elementer i diagrammet findes i næste kapitel.

Mønsterdele

Følgende er en liste over de dele, designmønstret består af.

Subject (subjekt) Kender de Observer-objekter (observatører) der er knyttet til objektet. Der kan være et vilkårligt antal Observer-objekter der overvåger et subjekt



Figur A.1: Strukturdiagram for observer-designmønstret.

(Subject). Desuden giver Subject et interface til at på- og afhægte Observer-objekter.

Observer Definerer et interface til at opdatere objekter der skal underrettes hvis der sker ændringer i et Subject.

ConcreteSubject Opbevarer tilstande, der kan have interesse for Observers og sender en besked til sine observatører når dens tilstand ændrer sig.

ConcreteObserver Opretholder en reference til et ConcreteSubject-objekt og gemmer tilstande, som skal være konsistente med subjekternes tilstande. Desuden implementerer ConcreteObserver den update-metode som Observer har, så ConcreteObservers tilstand hele tiden er konsistent med de tilhørende genstande.

Konsekvenser

At bruge mønstret har følgende fordele og ulemper:

1. Giver en abstrakt kobling mellem subjekter og observatører.
2. Understøtter muligheden for at kommunikere ved hjælp af broadcasting.
3. En mindre værdifuld konsekvens af mønstret er at det kan medføre mange opdateringer i observatørerne da observatør ikke har kendskab til andre observatører.

B Grammatik

pesto_file → *specialization_class**
main_file
(*extra_classes*)$^?$
(*entrypoint_overwrite*)$^?$

specialization_class → **specclass** *specclass_name* **specializes** *class_name* {
(*predicate_declaration*;)*
method_declaration$^?$
}

predicate_declaration → *variable_declaration* == *value_declaration*
| *variable_declaration* : *type_declaration*

method_declaration → *method_prototype* *parameter_declaration*

variable_declaration → *field_name*
| *parameter*

parameter_declaration → **where** *predicate_declaration*;
| {(**where** *predicate_declaration*;)$^+$}
| ;

value_declaration → *value*
| ?
| !

type_declaration → *array_type* ([*array_index*])$^+$(= *contents_declaration*)$^?$
| *types*

array_type → (*type* (| *type*)$^+$)
| *type*

array_index → *integer*
| ?
| !

types → *type* (| *type*)*

type → *specclass_name*
| *class_name*
| *type_name*

contents_declaration → [*types*]
| {*types* (, *types*)*}
| {*value* (, *value*)*}

specclass_name → *identifier*

main_file → **main file:** *identifier.java*

extra_classes → **extra classes:** {*identifier.java* (, *identifier.java*)*}

entrypoint_overwrite → **entrypoint overwrite:** (*binding_time* (, *binding_time*)*)

binding_time → S
| D

C Semantikken for Pesto

C.1 Analysekontekst

$lhs(C, v)$ = $inst\_C.v$

$lhs(\epsilon, v)$ = v

$S, J \in T$

$[\![E\ \ S_1\ \ S_2\ \ \ldots\ \ S_m]\!] =$

```
public class AnalysisContext {
    D[[E]]
    public static void setAnalysisContext() {
        I[[E]] I[[S1]] I[[S2]] ... I[[Sm]]
        [[E]] [[S1]] ... [[Sm]]
        _this = inst_E;
    }
}
```

$\mathcal{D}[\![$**specclass** E **specializes** $J\{\ p_1\ \ p_2\ \ \ldots p_k\ \ e\ \}]\!] =$

```
public static J _this;
D[[e]]
```

$\mathcal{D}[\![$**public** int $f($ $T_1\ \ p_1,\ \ T_2\ \ p_2,\ \ \ldots, T_h\ \ p_h)$ $\{$**where** $p_1\ \ \ldots$ **where** $p_k\ \}]\!] =$

```
public static T1 p1;
:
public static Th ph;
```

$\mathcal{I}[\![$**specclass** N **specializes** $J\{\ p_1\ \ p_2\ \ \ldots p_k\ \ e\ \}]\!] =$

```
J inst_N = new J();
```

$[\![$**specclass** E **specializes** $J\{\ p_1\ \ p_2\ \ \ldots p_k\ \ e\ \}]\!] =$

$[\![p_1]\!](E)\ [\![p_2]\!](E) \ldots [\![p_k]\!](E)\ [\![e]\!]$

$[\![$**specclass** S_i **specializes** $J_i\ \{\ p_1\ \ p_2\ \ \ldots p_k\ \}]\!] =$

$[\![p_1]\!](S_i)\ [\![p_2]\!](S_i) \ldots [\![p_k]\!](S_i)$

$[\![$**public** int $f(r_1,\ r_2, \ldots,\ r_h)$ $\{$**where** $p_1\ \ \ldots$ **where** $p_k\ \}]\!] =$

$[\![p_1]\!](\epsilon)\ [\![p_2]\!](\epsilon) \ldots [\![p_k]\!](\epsilon)$

$[\![v$ == ?$]\!](L) =$

```
lhs(L, v) = DynamicValue.get_int();
```

$[\![v \texttt{ == } \texttt{!}]\!](L) =$

$lhs(L, v)$ = StaticValue.get_int( );

$[\![v \texttt{ == } number]\!](L) =$

$lhs(L, v)$ = StaticValue.get_int( );

$[\![d]\!](L) =$

$lhs(L, d)$ = DynamicValue.get_int( );
for enhver primitiv instansvariabel d i Ls rodklassen som ikke er brugt i L

$[\![v : J]\!](L) =$

$lhs(L, v)$ = **new** J( );

$[\![v : S]\!](L) =$

$lhs(L, v) = inst\_S$;

$[\![v\texttt{: } T_1 \texttt{ | } T_2 \texttt{ | } \ldots \texttt{ | } T_k \texttt{;}]\!](L) =$

if (StaticValue.get_boolean( )) {
  $[\![v\texttt{: } T_1]\!](L)$
} **else** {
  $[\![v\texttt{: } T_2 \texttt{ | } T_3 \texttt{ | } \ldots \texttt{ | } T_k]\!](L)$
}

$[\![d]\!](L) =$

$lhs(L, d)$ = DynamicValue.get_object( );

for enhver instansvariabel d af en ikke-primitiv type i Ls rodklassen som ikke er brugt i L

C.2 Specialiseringskontekst

$lhs(C, v) =$ *inst_C*.v
$lhs(\epsilon, v) = v$
$vn(m, v) = m($$v$$)$
$vn(\epsilon, v) = v$
$str(C, v) =$ *s_C_v*
$str(\epsilon, v) =$ *s_v*
$S, J \in T$

$[\![E\ \ S_1\ \ S_2\ \ \ldots\ \ S_m]\!] =$

```
public class SpecializationContext {
    [[E]] [[S1]] ... [[Sm]]
}
```

$[\![$**specclass** E **specializes** $J\{\ p_1\ \ p_2\ \ \ldots p_k\ \ e\ \}]\!] =$

```
public static J _this;
D[[e]]
public static void setSpecializationContext() {
    guard = new StringBuffer();
    _this = meth_E(0);
    writeGuard(guard);
}
public static J meth_E(int n) {
    J inst_E = new J();
    S[[p1]](E,ε) S[[p2]](E,ε) ... S[[pk]](E,ε) S[[e]](ε,ε)
    guard_E(F[[p1]](E) ,..., F[[pk]](E) , F[[e]](ε) , n);
    A[[p1]](E) A[[p2]](E) ... A[[pk]](E) A[[e]](ε)
    return inst_E;
}
public static BitSet guardGenerated_E = new BitSet();
public static void guard_E(P[[p1]](E) , ... , P[[pk]](E) , P[[e]](ε) , int n) {
    if ( !guardGenerated_E.get(n)) {
        guard.append("public aspect Guard {\n");
        guard.append("public boolean J.guard_E(");
        guard.append(H[[e]] );
        guard.append(") {\n"
        G[[p1]](this.) G[[p2]](this.) ... G[[pk]](this.) G[[e]](ε)
        guardLine("return true;");
        guardLine("}");
        guardLine("pointcut entrypoint (J _j, P[[e]](ε) ):");
```

$\mathcal{C}[\![e]\!](J)$;
$\mathcal{O}[\![e]\!](J, E)$
guardLine("}");
guardGenerated_E.set(n);
}
}

$\mathcal{D}[\![$**public** int f(**int** r_1, **int** r_2, ..., **int** r_h) {**where** p_1 ...**where** p_k }$]\!] =$

public static int r_1;
public static int r_2;
⋮
public static int r_h;

$[\![$**specclass** S_i **specializes** J_i { p_1 p_2 ... p_k e }$]\!] =$

public static J meth_S_i() {
J_i *inst_*S_i = **new** J_i();
$\mathcal{S}[\![p_1]\!](S\_i, \epsilon)$ $\mathcal{S}[\![p_2]\!](S\_i, \epsilon)$ ... $\mathcal{S}[\![p_k]\!](S\_i, \epsilon)$
*guard_*S_i($\mathcal{F}[\![p_1]\!](E)$,..., $\mathcal{F}[\![p_k]\!](E)$, $\mathcal{F}[\![e]\!](\epsilon)$, n);
$\mathcal{A}[\![p_1]\!](S\_i)$ $\mathcal{A}[\![p_2]\!](S\_i)$ ... $\mathcal{A}[\![p_k]\!](S\_i)$
return *inst_*S_i;
}
public static BitSet *guardGenerated_*E = **new** BitSet();
public static void guard_S_i($\mathcal{P}[\![p_1]\!]$, ... , $\mathcal{P}[\![p_h]\!]$, int n) {
if (!*guardGenerated_*E.get(n)) {
guardLine("public boolean J_i.guard_S_i"+n+"() {");
$\mathcal{G}[\![p_1]\!]$(this.) $\mathcal{G}[\![p_2]\!]$(this.) ... $\mathcal{G}[\![p_k]\!]$(this.) $\mathcal{G}[\![e]\!](\epsilon)$
guardLine("return true;");
guardLine("}");
}
}

$\mathcal{S}[\![v$ == ?$]\!](C, m) =$

ϵ

$\mathcal{S}[\![v$ == !$]\!](C, m) =$

String $str(C, v)$ = getValue("C", "$vn(m, v)$", n);

$\mathcal{S}[\![v : J]\!](C, m) =$

ObjectValue $str(C, v)$ = **new** ObjectValue("J", n);

$\mathcal{S}[\![v : S]\!](C, m) =$

ObjectValue $str(C, v)$ = getTypeValue("C", "$vn(m, v)$", n);
if (noFile) { $str(C, v)$ = **new** ObjectValue("R", n);}

$\mathcal{S}[\![v$: $T_1 \mid T_2 \mid \ldots \mid T_k$;$]\!](C, m) =$

ObjectValue $str(C, v)$ = getTypeValue("C", "$vn(m, v)$", n);

$\mathcal{S}[\![\textbf{public}\ \mathsf{int}\ f(T_1\ r_1,\ T_2\ r_2,\ \ldots,T_h\ r_h)\ \{\textbf{where}\ p_1\ \ldots\textbf{where}\ p_k\ \}]\!](C,m) =$
$\quad \mathcal{S}[\![p_1]\!](C,f)\ \mathcal{S}[\![p_2]\!](C,f)\ \ldots\ \mathcal{S}[\![p_k]\!](C,f)$

$\mathcal{F}[\![v\ \ predicate]\!](C) =$
$\quad str(C,v)$

$\mathcal{F}[\![\textbf{public}\ \mathsf{int}\ f(T_1\ r_1,\ T_2\ r_2,\ \ldots,T_h\ r_h)\ \{\textbf{where}\ p_1\ \ldots\textbf{where}\ p_k\ \}]\!](C) =$
$\quad \mathcal{F}[\![p_1]\!](\epsilon)\ \mathcal{F}[\![p_2]\!](\epsilon)\ldots\mathcal{F}[\![p_k]\!](\epsilon)$

$\mathcal{A}[\![v\ \texttt{==}\ \texttt{?}]\!](C) =$
$\quad \epsilon$

$\mathcal{A}[\![v\ \texttt{==}\ \texttt{!}]\!](C) =$
$\quad lhs(C,v)$ = (Integer.valueOf)($str(C,v)$).intValue();

$\mathcal{A}[\![v\ \texttt{==}\ number]\!](C) =$
$\quad lhs(C,v)$ = $number$;

$\mathcal{A}[\![v : J]\!](C) =$
$\quad lhs(C,v)$ = **new** J();

$\mathcal{A}[\![v : S]\!](C) =$
$\quad lhs(C,v)$ = meth_S($str(C,v).n$);

$\mathcal{A}[\![v\texttt{:}\ \ T_1\,\texttt{|}\,T_2\,\texttt{|}\,\ldots\,\texttt{|}\,T_n\,\texttt{;}]\!](C) =$
$\quad$ **if** ($str(C,v).value$.equals("T_1")) {
$\quad\quad \mathcal{A}[\![v : T_1]\!](C)$;
$\quad$ } **else** {
$\quad\quad \mathcal{A}[\![v : T_2 \mid \ldots \mid T_n]\!](C)$
$\quad$ }

$\mathcal{A}[\![\textbf{public}\ \mathsf{int}\ f(T_1\ r_1,\ T_2\ r_2,\ \ldots,T_h\ r_h)\ \{\textbf{where}\ p_1\ \ldots\textbf{where}\ p_k\ \}]\!](C) =$
$\quad \mathcal{A}[\![p_1]\!](\epsilon)\ \mathcal{A}[\![p_2]\!](\epsilon)\ldots\mathcal{A}[\![p_k]\!](\epsilon)$

$\mathcal{P}[\![v\ \ :\ \ T]\!](C) =$
$\quad$ ObjectValue $str(C,v)$

$\mathcal{P}[\![v\ \texttt{==}\ \ldots]\!](C) =$
$\quad$ String $str(C,v)$

$\mathcal{P}[\![\textbf{public}\ \mathsf{int}\ f(T_1\ r_1,\ T_2\ r_2,\ \ldots,T_h\ r_h)\ \{\textbf{where}\ p_1\ \ldots\textbf{where}\ p_k\ \}]\!](C) =$
$\quad \mathcal{P}[\![p_1]\!]\epsilon\ \mathcal{P}[\![p_2]\!]\epsilon\ldots\mathcal{P}[\![p_k]\!]\epsilon$

$\mathcal{H}[\![\textbf{public}\ \mathsf{int}\ f(T_1\ r_1,\ T_2\ r_2,\ \ldots,T_h\ r_h)\ \{\textbf{where}\ p_1\ \ldots\textbf{where}\ p_k\ \}]\!] =$
$\quad T_1\ r_1, T_2\ r_2, \ldots, T_h\ r_h$

$\mathcal{G}[\![v\ \texttt{==}\ number]\!](p) =$
$\quad guardLine$("if (!(pv == $number$)) return false;");

$\mathcal{G}[\![v\texttt{:}\ J]\!](p) =$
$\quad guardLine$("if (!(pv.getClass() == J.class)) return false;");

$\mathcal{G}[\![v : S]\!](p) =$

guardLine(“if (!(pv.getClass() == R_S.class)) return false;”);
guardLine(“if (!((R_S)pv)._guard_S())) return false;”);

hvor $R_S = rootclass(S)$

$\mathcal{G}[\![v : \ T_1 \mid T_2 \mid \ldots \mid T_n ;]\!](p) =$

if (*str*(C, v).*value*.equals(“T_1”)) {
  $\mathcal{G}[\![v : T_1]\!](p)$;
} else {
  $\mathcal{G}[\![v : T_2 \mid \ldots \mid T_n]\!](p)$
}

$\mathcal{G}[\![$**public** int f(**T_1 r_1, T_2 r_2, ..., T_h r_h**) { where** p_1 **... where** p_l**}**$]\!](p) =$
$\mathcal{G}[\![p_1]\!](p)\ \mathcal{G}[\![p_2]\!](p)\ \ldots \mathcal{G}[\![p_k]\!](p)$

$\mathcal{C}[\![$**public** int f(**T_1 r_1, T_2 r_2, ..., T_h r_h**) {where** p_1 **... where** p_k**}**$]\!](R) =$
guardLine(“call (int R.f(T_1, T_2, ..., T_h))”);
guardLine(“&& args(r_1, r_2, ...,r_h)”);
guardLine(“&& target (r);”);

$\mathcal{O}[\![$**public** int f(**T_1 r_1, T_2 r_2, ..., T_h r_h**) {where** p_1 **... where** p_k**}**$]\!](R, L) =$
guardLine(“int around(R r, T_1 r_1, T_2 r_2, ..., T_h r_h):”);
guardLine(“entrypoint (r, r_1, r_2, ..., r_h) {”);
guardLine(“if (r.guard_L(r_1, r_2, ...,r_h)) {”);
guardLine(“return j.f_spec(r_1, r_2, ...,r_h);”);
guardLine(“} else }”);
guardLine(“return proceed (R, r_1, r_2, ..., r_h);”);
guardLine(“}”);
guardLine(“}”);

hvor $r = lowercase(R)$ og $j = lowercase(J)$

D JSpec-kontekst til beregning af udtryk

Følgende er indeholdet af _JSpec_Context til AnyExpression som den bliver genereret af Pesto. Jeg har kortet i filen i et forsøg på at gøre den mere overskuelig. Det kode jeg har klippet ud er typisk skrevet efter samme skabelon som det forrige i en tilsvarende metode og der er derfor ikke gået noget tabt ved disse udklip.

Sidst i filen ses de metoder setSpecializationContext bruger til at læse fra værdifilen. Vogteren der bliver genereret af dette kode og værdifilen i figur 4.7 er den der ses i figur 5.23.

```
/* This file is generated automatic by Specialization Classes */
import fr.irisa.compose.jspec.StaticValue;
import fr.irisa.compose.jspec.DynamicValue;
import java.util.*;
import java.io.*;
public class _JSpec_Context {
  public static Calculate _this;
  public static Exp e;
  public static float[] env;
  public static File valuesTemplate;
  public static BufferedInputStream template;
  public static File guardFile;
  public static DataOutputStream guardWriter;
  public static StringBuffer guard;
  public static StringBuffer pointcut;
  public static boolean noFile;
  public static void main(String[] args) {
    Main.main(args);
  }
  public static void setAnalysisContext() {
    Calculate inst_AnyExpression = new Calculate();
    Multiplication inst_SpecMult = new Multiplication();
    Addition inst_SpecAdd = new Addition();
    Negation inst_SpecNeg = new Negation();
    Constant inst_SpecConst = new Constant();
    Variable inst_SpecVar = new Variable();
    // Predicates in AnyExpression
    if (StaticValue.get_boolean()) {
```

```
    e = inst_SpecMult;
  } else {
    if (StaticValue.get_boolean()) {
      e = inst_SpecAdd;
    } else {
      if (StaticValue.get_boolean()) {
        e = inst_SpecConst;
      } else {
        if (StaticValue.get_boolean()) {
          e = inst_SpecVar;
        } else {
          e = inst_SpecNeg;
        }
      }
    }
  }
  env = new float[StaticValue.get_int()];
  env[StaticValue.get_int()] = DynamicValue.get_float();
  // Predicates in SpecMult
  if (StaticValue.get_boolean()) {
    inst_SpecMult.left = inst_SpecMult;
  } else {
    if (StaticValue.get_boolean()) {
      inst_SpecMult.left = inst_SpecAdd;
    } else {
      if (StaticValue.get_boolean()) {
        inst_SpecMult.left = inst_SpecConst;
      } else {
        if (StaticValue.get_boolean()) {
          inst_SpecMult.left = inst_SpecVar;
        } else {
          inst_SpecMult.left = inst_SpecNeg;
        }
      }
    }
  }
  if (StaticValue.get_boolean()) {
    inst_SpecMult.right = inst_SpecMult;
  } else {
    ...
    // kode som for inst_SpecMult.left
  }
  // Predicates in SpecAdd
  if (StaticValue.get_boolean()) {
    inst_SpecAdd.left = inst_SpecMult;
  } else {
    ...
    // kode som for inst_SpecMult.left
  }
```

```
    if (StaticValue.get_boolean()) {
      inst_SpecAdd.right = inst_SpecMult;
    } else {
      ...
      // kode som for inst_SpecMult.left
    }
    // Predicates in SpecNeg
    if (StaticValue.get_boolean()) {
      inst_SpecNeg.arg = inst_SpecMult;
    } else {
      ...
      // kode som for inst_SpecMult.left
    }
    // Predicates in SpecConst
    inst_SpecConst.value = StaticValue.get_float();
    // Predicates in SpecVar
    inst_SpecVar.identifier = StaticValue.get_int();
    // Unused variables in AnyExpression
    // Unused variables in SpecMult
    // Unused variables in SpecAdd
    // Unused variables in SpecNeg
    // Unused variables in SpecConst
    // Unused variables in SpecVar
    _this = inst_AnyExpression;
  }
  public static void setSpecializationContext() {
    guard = new StringBuffer();
    _this = meth_AnyExpression(0);
    writeGuard();
    try {
      // Writing to the guard aspect file
      guardWriter.writeBytes("public aspect Guard{\n\n");
      guardWriter.writeBytes(guard.toString());
      guardWriter.writeBytes(pointcut.toString());
      guardWriter.close();
    } catch (IOException e) {
      System.err.println(e);
      System.exit(0);
    }
  }
  public static Calculate meth_AnyExpression(int n) {
    Calculate inst_AnyExpression = new Calculate();
    ObjectValue s_e = getTypeValue("AnyExpression", "calc(e)", n);
    ObjectValue s_env = new ObjectValue("float", new int[] {2},
                                        new String[] {}, n);
    guard_AnyExpression(s_e, s_env, n);
    if (s_e.value.equals("SpecMult")) {
      e = meth_SpecMult(s_e.n);
```

```
    } else {
      if (s_e.value.equals("SpecAdd")) {
        e = meth_SpecAdd(s_e.n);
      } else {
        if (s_e.value.equals("SpecConst")) {
          e = meth_SpecConst(s_e.n);
        } else {
          if (s_e.value.equals("SpecVar")) {
            e = meth_SpecVar(s_e.n);
          } else {
            e = meth_SpecNeg(s_e.n);
          }
        }
      }
    }
    env = new float[2];
    return inst_AnyExpression;
  }
  public static BitSet guard_generated_AnyExpression = new BitSet();
  public static void guard_AnyExpression(ObjectValue _pesto_e,
                                         ObjectValue _pesto_env, int n) {
    if (!guard_generated_AnyExpression.get(n)) {
      guard.append(" private boolean Calculate._guard_AnyExpression(");
      guard.append("Exp e, float[] env");
      guard.append("){\n");
      if (_pesto_e.value.equals("SpecMult")) {
        guard.append("   if (!(e.getClass() == Multiplication.class))");
        guard.append("return false;\n");
        guard.append("   if (!(((Multiplication)e)._guard_SpecMult"+
                        _pesto_e.n+"())) return false;");
      } else {
        if (_pesto_e.value.equals("SpecAdd")) {
          guard.append("   if (!(e.getClass() == Addition.class))");
          guard.append("return false;\n");
          guard.append("   if (!(((Addition)e)._guard_SpecAdd"+
                          _pesto_e.n+"())) return false;\n");
        } else {
          if (_pesto_e.value.equals("SpecConst")) {
            guard.append("   if (!(e.getClass() == Constant.class)) return false;\n");
            guard.append("   if (!(((Constant)e)._guard_SpecConst"+
                            _pesto_e.n+"())) return false;\n");
          } else {
            if (_pesto_e.value.equals("SpecVar")) {
              guard.append("   if (!(e.getClass() == Variable.class)) return false;\n");
              guard.append("   if (!(((Variable)e)._guard_SpecVar"+
                              _pesto_e.n+"())) return false;\n");
            } else {
              guard.append("   if (!(e.getClass() == Negation.class)) return false;\n");
```

```
                guard.append("    if (!(((Negation)e)._guard_SpecNeg"+
                                    _pesto_e.n+"())) return false;\n");
              }
            }
          }
        }
        guard.append("    if (!(env.getClass() == "+
                        _pesto_env.value+"[].class)) return false;\n");
        guard.append("    if (!(env.length == "+
                        _pesto_env.lengths[0]+")) return false;\n");
        guard.append("    return true;\n");
        guard.append("  }\n\n");
        guard_generated_AnyExpression.set(n);
      }
    }
    public static Multiplication meth_SpecMult(int n) {
      Multiplication inst_SpecMult = new Multiplication();
      ObjectValue s_this_left = getTypeValue("SpecMult", "left", n);
      ObjectValue s_this_right = getTypeValue("SpecMult", "right", n);
      guard_SpecMult(s_this_left, s_this_right, n);
      if (s_this_left.value.equals("SpecMult")) {
        inst_SpecMult.left = meth_SpecMult(s_this_left.n);
      } else {
        if (s_this_left.value.equals("SpecAdd")) {
          ...
        }
      }
      if (s_this_right.value.equals("SpecMult")) {
        inst_SpecMult.right = meth_SpecMult(s_this_right.n);
      } else {
        if (s_this_right.value.equals("SpecAdd")) {
          ...
        }
      }
      return inst_SpecMult;
    }
    public static BitSet guard_generated_SpecMult = new BitSet();
    public static void guard_SpecMult(ObjectValue _pesto_left,
                                      ObjectValue _pesto_right, int n) {
      if (!guard_generated_SpecMult.get(n)) {
        guard.append("  private boolean Multiplication._guard_SpecMult"+n+"(");
        guard.append("){\n");
        if (_pesto_left.value.equals("SpecMult")) {
          guard.append("    if (!(this.left.getClass() == Multiplication.class)) return false;\n");
          guard.append("    if (!(((Multiplication)this.left)._guard_SpecMult"+
                          _pesto_left.n+"())) return false;\n");
        } else {
          if (_pesto_left.value.equals("SpecAdd")) {
```

```
            ...
          }
        }
        if (_pesto_right.value.equals("SpecMult")) {
          guard.append("    if (!(this.right.getClass() == Multiplication.class)) return false;\n");
          guard.append("    if (!(((Multiplication)this.right)._guard_SpecMult"+
                                _pesto_right.n+"())) return false;\n");
        } else {
          if (_pesto_right.value.equals("SpecAdd")) {
            ...
          }
        }
        guard.append("    return true;\n");
        guard.append("  }\n\n");
        guard_generated_SpecMult.set(n);
      }
    }
    public static Addition meth_SpecAdd(int n) {
      Addition inst_SpecAdd = new Addition();
      ObjectValue s_this_left = getTypeValue("SpecAdd", "left", n);
      ObjectValue s_this_right = getTypeValue("SpecAdd", "right", n);
      guard_SpecAdd(s_this_left, s_this_right, n);
      if (s_this_left.value.equals("SpecMult")) {
        inst_SpecAdd.left = meth_SpecMult(s_this_left.n);
      } else {
        if (s_this_left.value.equals("SpecAdd")) {
          ...
        }
      }
      inst_SpecAdd.left = _pesto_tmp_left;
      if (s_this_right.value.equals("SpecMult")) {
        inst_SpecAdd.right = meth_SpecMult(s_this_right.n);
      } else {
        if (s_this_right.value.equals("SpecAdd")) {
          ...
        }
      }
      inst_SpecAdd.right = _pesto_tmp_right;
      return inst_SpecAdd;
    }
    public static BitSet guard_generated_SpecAdd = new BitSet();
    public static void guard_SpecAdd(ObjectValue _pesto_left,
                                     ObjectValue _pesto_right, int n) {
      if (!guard_generated_SpecAdd.get(n)) {
        guard.append("  private boolean Addition._guard_SpecAdd"+n+"(");
        guard.append("){\n");
        if (_pesto_left.value.equals("SpecMult")) {
          guard.append("    if (!(this.left.getClass() == Multiplication.class)) return false;\n");
```

```
        guard.append("    if (!(((Multiplication)this.left)._guard_SpecMult"+
                        _pesto_left.n+"())) return false;\n");
      } else {
        if (_pesto_left.value.equals("SpecAdd")) {
          ...
        }
      }
      if (_pesto_right.value.equals("SpecMult")) {
        guard.append("    if (!(this.right.getClass() == Multiplication.class)) return false;\n");
        guard.append("    if (!(((Multiplication)this.right)._guard_SpecMult"+
                        _pesto_right.n+"())) return false;\n");
      } else {
        if (_pesto_right.value.equals("SpecAdd")) {
          ...
        }
      }
      guard.append("    return true;\n");
      guard.append("  }\n\n");
      guard_generated_SpecAdd.set(n);
    }
  }
  public static Negation meth_SpecNeg(int n) {
    Negation inst_SpecNeg = new Negation();
    ObjectValue s_this_arg = getTypeValue("SpecNeg", "arg", n);
    guard_SpecNeg(s_this_arg, n);
    if (s_this_arg.value.equals("SpecMult")) {
      inst_SpecNeg.arg = meth_SpecMult(s_this_arg.n);
    } else {
      if (s_this_arg.value.equals("SpecAdd")) {
        ...
      }
    }
    return inst_SpecNeg;
  }
  public static BitSet guard_generated_SpecNeg = new BitSet();
  public static void guard_SpecNeg(ObjectValue _pesto_arg, int n) {
    if (!guard_generated_SpecNeg.get(n)) {
      guard.append("  private boolean Negation._guard_SpecNeg"+n+"(");
      guard.append("){\n");
      if (_pesto_arg.value.equals("SpecMult")) {
        guard.append("    if (!(this.arg.getClass() == Multiplication.class)) return false;\n");
        guard.append("    if (!(((Multiplication)this.arg)._guard_SpecMult"+
                        _pesto_arg.n+"())) return false;\n");
      } else {
        if (_pesto_arg.value.equals("SpecAdd")) {
          ...
        }
      }
```

```
      guard.append("    return true;\n");
      guard.append("  }\n\n");
      guard_generated_SpecNeg.set(n);
    }
  }
  public static Constant meth_SpecConst(int n) {
    Constant inst_SpecConst = new Constant();
    String s_this_value = "3.1416F";
    guard_SpecConst(s_this_value, n);
    inst_SpecConst.value = 3.1416F;
    return inst_SpecConst;
  }
  public static BitSet guard_generated_SpecConst = new BitSet();
  public static void guard_SpecConst(String _pesto_value, int n) {
    if (!guard_generated_SpecConst.get(n)) {
      guard.append("  private boolean Constant._guard_SpecConst"+n+"(");
      guard.append("){\n");
      guard.append("    if (!(this.value == "+
                    _pesto_value+")) return false;\n");
      guard.append("    return true;\n");
      guard.append("  }\n\n");
      guard_generated_SpecConst.set(n);
    }
  }
  public static Variable meth_SpecVar(int n) {
    Variable inst_SpecVar = new Variable();
    String s_this_identifier = getValue("SpecVar", "identifier", n);
    guard_SpecVar(s_this_identifier, n);
    inst_SpecVar.identifier = (Integer.valueOf(s_this_identifier)).intValue();
    return inst_SpecVar;
  }
  public static BitSet guard_generated_SpecVar = new BitSet();
  public static void guard_SpecVar(String _pesto_identifier, int n) {
    if (!guard_generated_SpecVar.get(n)) {
      guard.append("  private boolean Variable._guard_SpecVar"+n+"(");
      guard.append("){\n");
      guard.append("    if (!(this.identifier == "+
                    _pesto_identifier+")) return false;\n");
      guard.append("    return true;\n");
      guard.append("  }\n\n");
      guard_generated_SpecVar.set(n);
    }
  }
  private static class ObjectValue {
    String value;
    int[] lengths;
    ObjectValue[] contents;
    int n;
```

```
public ObjectValue(String value, int[] lengths, String[] contents, int n) {
  int index;
  this.value = value;
  this.lengths = lengths;
  this.contents = new ObjectValue[contents.length];
  for (int i = 0; i < contents.length; i++) {
    if ((index = contents[i].indexOf("#")) != -1) {
      this.contents[i] =
      new ObjectValue(contents[i].substring(0,index),
                      (Integer.valueOf(contents[i].substring(index+1))).intValue());
    } else {
      this.contents[i] = new ObjectValue(contents[i], 0);
    }
  }
  this.n = n;
}
public ObjectValue(String value, int[] lengths, ObjectValue[] contents, int n) {
  this.value = value;
  this.lengths = lengths;
  this.contents = contents;
  this.n = n;
}
public ObjectValue(String value, int[] lengths, int n) {
  this(value, lengths, new String[] {}, n);
}
public ObjectValue(String value, int n) {
  this(value, null, new String[] {}, n);
}
public String toString() {
  StringBuffer sb = new StringBuffer();
  sb.append("value: "+value);
  sb.append(", lengths: [");
  if (lengths != null) {
    for (int i = 0; i < lengths.length; i++) {
      sb.append(lengths[i]+", ");
    }
  } else {
    sb.append("null");
  }
  sb.append("]");
  sb.append(", contents: [");
  if (contents != null) {
    for (int i = 0; i < contents.length; i++) {
      sb.append(contents[i].toString()+", ");
    }
  } else {
    sb.append("null");
  }
```

```
      sb.append("]");
      sb.append(", n: "+n);
      return sb.toString();
    }
  }
  private static String getValue(String specclass, String field, int n) {
    String s = findField(specclass, field, "=", n); // the line with the field
    int index1 = s.indexOf("=");
    int index2 = s.indexOf(";");
    s = s.substring(index1+1, index2); // everything between 'type' and ';'
    s = removeBlanks(s);
    return s;
  }
  private static ObjectValue getTypeValue(String specclass, String field, int n) {
    ObjectValue ev;
    String s = findField(specclass, field, ":", n); // the line with the field
    if (noFile) {
      return null;
    } else {
      int index1 = s.indexOf(":");
      int index2 = s.indexOf(";");
      s = removeBlanks(s.substring(index1+1, index2)); // everything between 'type' and ';'
      index1 = s.indexOf("#");
      if (index1 != -1) {
        return ev =
        new ObjectValue(s.substring(0,index1),
                        (Integer.valueOf(s.substring(index1+1))).intValue());
      } else {
        return ev = new ObjectValue(s, 0);
      }
    }
  }
  private static ObjectValue getArray(String specclass, String field,
                                      String value, int[] lengths,
                                      String[] contents, int n) {
    String s = findField(specclass, field, ":", n);
    if (noFile) {
      return null;
    } else {
      int index1 = s.indexOf(":");
      int index2 = s.indexOf(";");
      s = s.substring(index1+1, index2); // everything between 'type' and ';'
      // get the value before the brackets: v:(A|B)[];
      if (value == null) {
        value = readArrayType(s);
      }
      // get the lengths v: ][int][int]...[int]
      if (lengths == null) {
```

```
        lengths = readArrayLengths(s);
      }
      if (contents == null) {
        contents = readArrayContents(s);
      }
      return new ObjectValue(value, lengths, contents, n);
    }
  }
  private static String[] readArrayContents(String s) {
    String[] list;
    int index1 = s.indexOf("="), index2;
    s = s.substring(index1);
    if ((index1 = s.indexOf("{")) != -1) {
      int dim = getContentsLength(s);
      list = new String[dim];
      int i = 0;
      index1++;
      while ((index2 = s.indexOf(",")) != -1) {
        list[i] = removeBlanks(s.substring(index1,index2));
        i++;
        s = s.substring(index2+1);
        index1 = 0;
      }
      list[i] = removeBlanks(s.substring(index1,s.indexOf("}")));
    } else {
      if ((index1 = s.indexOf("[")) != -1) {
        list = new String[1];
        index2 = s.indexOf("]");
        list[0] = removeBlanks(s.substring(index1+1, index2));
      } else {
        list = new String[0];
      }
    }
    return list;
  }
  private static int getContentsLength(String s) {
    int i = 0, index;
    while ((index = s.indexOf(",")) != -1) {
      i++;
      s = s.substring(index+1);
    }
    return ++i;
  }
  private static String readArrayType(String s) {
    int index1 = s.indexOf("(");
    int index2 = s.indexOf(")");
    if (s.indexOf("|") != -1) {
      System.err.println("Error in the values file. Edit the array type to contain one type");
```

```
      System.exit(2);
    }
    s = removeBlanks(s.substring(index1+1, index2));
    return s;
  }
  private static int[] readArrayLengths(String s){
    int index1, index2;
    int[] list; int dim;
    if ((index1 = s.indexOf("=")) != -1) { // the contents declaration is removed
      s = s.substring(0,index1);
    }
    dim = getDimension(s);
    list = new int[dim];
    for (int i = 0; i < dim; i++) {
      index1 = s.indexOf("[");
      index2 = s.indexOf("]");
      list[i] =
      ((Integer.valueOf(removeBlanks(s.substring(index1+1, index2)))).intValue());
      s = s.substring(index2+1);
    }
    return list;
  }
  private static int getDimension(String s) {
    int d = 0, index;
    while ((index = s.indexOf("[")) != -1) {
      s = s.substring(index+1);
      d++;
    }
    return d;
  }
  private static String removeBlanks(String valueString) {
    String s = valueString;
    while (s.startsWith(" ")) {
      s = s.substring(1);
    }
    while (s.endsWith(" ") || s.endsWith("\t")) {
      s = s.substring(0,s.length()-1);
    }
    return s;
  }
  private static String findField(String specclass, String field, String type, int n) {
    String s = "";
    try {
      valuesTemplate = new File("final.values");
      template = new BufferedInputStream(new FileInputStream(valuesTemplate));
      int length = template.available();
      byte[] b = new byte[length];
      template.read(b, 0, length);
```

```
        s = new String(b, 0);
        if (n>0) {
          specclass = specclass+"#"+n; }
        int index1 = s.indexOf(specclass+" '");
        int index2 = s.indexOf("'", index1);
        s = s.substring(index1, index2);
        index1 = s.indexOf(" "+field+" "+type);
        index2 = s.indexOf("\n", index1);
        s = s.substring(index1, index2);
      } catch (FileNotFoundException e) {
        noFile = true;
      } catch (IOException e1) {
        System.err.println(e1);
        System.exit(2);
      }
      return s;
    }
    private static void writeGuard() {
      pointcut = new StringBuffer();
      try {
        guardFile = new File("Guard.java");
        guardWriter = new DataOutputStream(new FileOutputStream(guardFile));
      } catch (FileNotFoundException e1) {
        System.err.println(e1);
        System.exit(2);
      } catch (IOException e2) {
        System.err.println(e2);
        System.exit(3);
      }
      pointcut.append("  pointcut _calc_entrypoint ");
      pointcut.append("(Calculate _calculate, Exp _calc_e, float[] _calc_env):\n");
      pointcut.append("    call (float Calculate.calc(Exp, float[]))\n");
      pointcut.append("    && args(_calc_e, _calc_env)\n");
      pointcut.append("    && target(_calculate);\n\n");
      pointcut.append("  float around(Calculate _calculate, Exp _calc_e, float[] _calc_env):\n");
      pointcut.append("    _calc_entrypoint(_calculate, _calc_e, _calc_env) {\n");
      pointcut.append("    if (_calculate._guard_AnyExpression(_calc_e, _calc_env)) {\n");
      pointcut.append("      return _calculate._calc_spec(_calc_e, _calc_env);\n");
      pointcut.append("    } else {\n");
      pointcut.append("      return proceed(_calculate, _calc_e, _calc_env);\n");
      pointcut.append("    }\n  }\n}\n");
    }
}
```

E Kode til den binære eksponentionfunktion

BinOp.java

```
public class BinPower {
   int exp;
   BinOp op;

   public BinPower( ){
   }

   public BinPower(int exp, BinOp op) {
      this.exp = exp;
      this.op = op;
   }

   public int raise(int base) {
      int result = this.op.neutral;
      int e = this.exp;
      while(e - - > 0) {
         result = op.apply(result, base);
      }
      return result;
   }
}
```

Add.java

```
public class Add extends BinOp {
   public Add( ) {
      this.neutral = 0;
   }

   public int apply(int result, int base) {
      return result + base;
   }
}
```

Mult.java

```
public class Mult extends BinOp{
   public Mult( ) {
      this.neutral = 1;
   }

   public int apply(int result, int base) {
      return result ∗ base;
   }
}
```

BinPower.java

```
public class BinPower {
   int exp;
   BinOp op;

   public BinPower( ){
   }

   public BinPower(int exp, BinOp op) {
      this.exp = exp;
      this.op = op;
   }

   public int raise(int base) {
      int result = this.op.neutral;
      int e = this.exp;
      while(e - - > 0) {
         result = op.apply(result, base);
      }
      return result;
   }
}
```

Litteratur

[Chambers93] C. Chambers. **Predicate Classes**. I O. Nierstrasz (red.), *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93)*, bind 707 af *Lecture Notes in Computer Science*, side 268–296, Kaiserslautern, Tyskland, juli 1993. Springer-Verlag.

[Consel93] C. Consel og O. Danvy. **Tutorial Notes on Partial Evaluation**. I *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'93)*, side 493–501, Charleston, South Carolina, USA, januar 1993. ACM Press.

[Consel96] C. Consel, L. Hornof, F. Noël, J. Noyé og E.-N. Volanschi. **A uniform approach for compile-time and run-time specialization**. I O. Danvy, R. Gluck og P. Thiemann (red.), *Partial Evaluation. International Seminar*, bind 1110 af *Lecture Notes in Computer Science*, side 54–72, Dagstuhl Castle, Tyskland, februar 1996. Springer-Verlag, Berlin.

[Cowan96] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel og E.-N. Volanschi. **Specialization Classes: An Object Framework for Specialization**. I *Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, Seattle, Washington, USA, oktober 1996.

[Deransart87] P. Deransart og G. Ferrand. **An operational formal definition of Prolog**. I *Proceedings of the 1987 Symposium on Logic Programming*, side 162–172, San Francisco, Californien, USA, august 1987. IEEE Computer Society Press.

[Deursen00] A. van Deursen, P. Klint og J. Visser. **Domain-specific languages: an annotated bibliograpgy**. *ACM SIGPLAN Notices*, 35(6), side26–36, juni 2000.

[Flanagan97] D. Flanagan. **Java in a nutshell – A desktop quick reference**. O'Reilly, anden udgave, 1997.

[Gagnon98] E. Gagnon. **SableCC, An Object-Oriented Compiler Framework**. Speciale, School of Computer Science McGill University, Montreal, Canada, marts 1998.

[Gamma95] E. Gamma, R. Helm, R. Johnson og J. Vlissides. **Design patterns – Elements of resuable object-oriented software**. Professional computing series. Addison-Wessley, 1995.

[Goguen88] J. Goguen og T. Winkler. **Introducing OBJ3**. Teknisk rapport SRI-CSL-88-9, SRI International, Computer Science Lab, august 1988.

[Jensen75] K. Jensen og N. Wirth. **Pascal User Manual and Report**. Springer-Verlag, anden udgave, 1975.

[Jones93] N. D. Jones, C. K. Gomard og P. Sestoft. **Partial Evaluation and Automatic Program Generation**. International Series in Computer Science. Prentice-Hall, juni 1993.

[Kernighan88] B. W. Kernighan og D. M. Ritchie. **The C programming language**. Software Series. Prentice-Hall, anden udgave, 1988.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier og J. Irwin. **Aspect-Oriented Programming**. I M. Aksit og S. Matsuoka (red.), *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, bind 1241 af *Lecture Notes in Computer Science*, side 220–242, Jyväskylä, Finland, juni 1997. Springer-Verlag.

[Kiczales01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm og W. G. Griswold. **An Overview of AspectJ**. I J. L. Knudsen (red.), *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, bind 2072 af *Lecture Notes in Computer Science*, side 327–353, Budapest, Ungarn, juni 2001. Springer-Verlag.

[Lawall99] J. L. Lawall og G. Muller. **Efficient Incremental Checkpointing of Java Programs**. *IRISA-99-1264*, 1999.

[Madsen93] O. L. Madsen, B. Møller-Pedersen og K. Nygaard. **Object-oriented programming in the Beta programming Language**. ACM Press, 1993.

[McCarthy62] J. McCarthy. **Lisp 1.5 Programmer's Manual**. MIT Press, 1962.

[Meur02a] A.-F. L. Meur, C. Consel og B. Escrig. **An environment for building cusomizable components**. I *IFIP/ACM Conference on Component Deployment*, Berlin, Tyskland, juni 2002.

[Meur02b] A.-F. L. Meur, J. L. Lawall og C. Consel. **Bridging the Gap Between Programming Languages and Partial Evaluation**. I *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, side 9–18, Portland, Oregon, USA, januar 2002. ACM SIGPLAN Notices.

[Milner97] R. Milner, M. Tofte, R. Harper og D. B. MacQueen. **The Definition of Standard ML (Revised)**. MIT Press, USA, 1997.

[Muller99] G. Muller og U. P. Schultz. **Harissa: A Hybrid Approach to Java Execution**. *IEEE Software*, 16(2), side44–51, 1999.

[Navarro94] J. J. M. Navarro. **Expressivity of Functional-logic Languages and their Implementation**. I M. Alpuente og R. Barbuti (red.), *Joint Conference on Declarative Programming GULP-PRODE'94*. GULP (Italian ALP Chapter), Universidad Politécnica Valencia, Servicio de publicaciones Universidad Politécnica de Valencia, september 1994.

[Paine95] J. Paine. **Introduction to Prolog for Mathmaticians**. 1995.
http://burks.brighton.ac.uk/burks/language/prolog/pms/pms.htm

[Ralston00] A. Ralston, E. D. Reilly og D. Hemmendinger (red.). **Encyclopedia of Computer Science**. Nature Publishing Group, 4. udgave, 2000.

[Schmidt95] E. M. Schmidt og M. I. Schwartzbach. **Programmering og programmeringssproget TRINE**. Datalogisk Afdeling, Århus, Danmark, 1995.

[Schultz99] U. P. Schultz, J. L. Lawall, C. Consel og G. Muller. **Towards Automatic Specialization of Java Programs**. I R. Guerraoui (red.), *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, bind 1628 af *Lecture Notes in Computer Science*, side 367–390, Lisabon, Portugal, juni 1999. Springer-Verlag.

[Schultz00] U. P. Schultz, J. L. Lawall, C. Consel og G. Muller. **Specialization Patterns**. I *The 15th IEEE International Conference on Automated Software Engineering*, side 197–206, Grenoble, Frankrig, september 2000.

[Schultz02] U. P. Schultz, J. L. Lawall og C. Consel. **Automatic Program Specialization for Java**. DAIMI-PB 551, 2002.

[Sprognævnet03] Sprognævnet. **Dansk Sprognævns hjemmeside**. 2003.
http://www.dsn.dk/

[tec88] **TechKnowledge Corp.** 1988.
http://www.techknowledgecorp.com/

[Tip99] F. Tip, C. Laffra, P. F. Sweeney og D. Streeter. **Practical experience with an application extractor for Java**. I *OOPSLA'99 Conference Proceedings*, side 292–305, Denver, Colorado, USA, november 1999. ACM Press.

[Volanschi97] E.-N. Volanschi, C. Consel, G. Muller og C. Cowan. **Declarative Specialization of Object-Oriented Programming**. I *OOPSLA'97 Conference Proceedings*, side 286–300, Atlanta, Georgia, USA, oktober 1997. ACM Press.

[Volanschi98a] E.-N. Volanschi. **An automatic approach to specializing system components**, juni 1998. Thesis outline.

[Volanschi98b] E.-N. Volanschi. **Une approche automatique à la spécialisation de composants système**. Ph.d-afhandling, Université de Rennes I, Frankrig, februar 1998.