UNIVERSITETET I OSLO
Institutt for informatikk

BooPT: Implementasjon av Package Templates for Boo

Masteroppgave

Håkon Stordahl

19. desember 2011

Forord

Takk til mine veiledere under denne oppgaven, Stein Krogdahl og Eyvind W. Axelsen, for mange gode råd og mye hjelp under mitt arbeid med denne oppgaven.

Innhold

1 Innledning **4**

2 Package Templates **8**

2.1 Omnavning 10
2.2 Utvidelse av klasser ved instansiering (adds-klasser) 10
2.3 Bevaring av klassehierarkier 12
2.4 Flere instansieringer av samme template 12
2.5 Klassesammenslåing 12
2.6 Typeparametrisering 14
2.7 Template-hierarkier 14

3 Boo **16**

3.1 Kompilering og kjøring av Boo-programmer 20
3.2 Kompilatorarkitektur 21
3.2.1 Kompilatorsekvensen 22
3.2.2 Det abstrakte syntakstreet 24
3.2.3 Predefinerte kompilatorsekvenser 28
3.2.4 Egendefinerte kompilatorsekvenser 28
3.3 Metaprogrammering 30
3.3.1 Metametoder 31
3.3.2 Makroer 34
3.3.3 Syntaktiske attributter 37

4 Boo Package Templates **41**

4.1 Pakker i Boo 42

4.2 Bruk av BooPT . . . 42
4.2.1 Deklarasjon av templater . . . 43
4.2.2 Instansiering av templater . . . 43
4.3 Implementasjon . . . 46
4.3.1 `template` . . . 47
4.3.2 `adds` . . . 50
4.3.3 `inst` . . . 51
4.4 Avsluttende vurderinger . . . 58
4.4.1 Syntaks . . . 59
4.4.2 Format for kompilert template . . . 61
4.4.3 Organisering av implementasjonen . . . 63

5 Status for BooPT og videre arbeid 65
5.1 PT-egenskaper som ikke er implementert . . . 65
5.2 Ekstra kompilatorsteg . . . 66
5.3 Syntaks . . . 66
5.4 Testing . . . 67
5.5 Konklusjon . . . 67

A Kildekode for BooPT 70
A.1 `TemplateMacro` . . . 70
A.2 `InstMacro.boo` . . . 72
A.3 `AddsAttribute.boo` . . . 82
A.4 `CompileWithTranslateGeneratedNamesStep.boo` . . . 83

Kapittel 1

Innledning

Dette er resultatet av arbeidet med en kort masteroppgave som i henhold til reglene har blitt gjennomført over en periode på 17 uker. Oppgaven som ble gitt var følgende:

> Sett deg inn i programmeringsspråket Boo, med spesiell vekt på dets mekanismer for å kunne utvide språket selv. Vurder i hvilken grad disse mekanismene er kraftige nok til å implementere en variant av Package Templates for Boo. Om mulig, lag en pilot-implementasjon.

Både *Package Templates* (PT)[1] og Boo[2] er forholdsvis nye språk og antageligvis ukjente for mange. PT er en mekanisme for gjenbruk av kode og Boo er et generelt programmeringsspråk som har gode egenskaper for såkalt metaprogrammering. I denne oppgaven har jeg altså sett på hvor kraftige disse egenskapene til Boo er og hvordan de egner seg for å implementere en variant av PT.

Package Templates er en mekanisme for å tilrettelegge, og utvide mulighetene, for gjenbruk av samlinger av klasser. Denne teknikken ble først foreslått i 2009 i en artikkel av Stein Krogdahl, Birger Møller-Pedersen og Fredrik Sørensen[1].

De fleste objektorienterte programmeringsspråk har mekanismer for å samle klasser i avgrensede enheter. Disse kalles for eksempel pakker (packages) i Java, navnerom i C# eller moduler i Python. Jeg vil bruke ordet *pakke* siden det er den betegnelsen som brukes i PT.

Slike pakker brukes gjerne til å samle klasser som på en eller annen måte hører sammen, eller i det minste er relatert. En slik inndeling i pakker er nyttig av to grunner. De kan bidra til bedre organisering av kode i store programmer, og kan dermed gi mer oversiktlige programmer. Dessuten

kan pakker utgjøre passelige enheter for gjenbruk av kode, for eksempel i programbiblioteker.

Pakker er altså en form for modularisering, litt på samme måte som inndeling av kode i funksjoner og klasser. I tillegg til å understøtte modularisering, kan pakker brukes til å regulere synligheten av innholdet i pakken. I PT er det spesielt gjenbruk av kode det fokuseres på.

Pakker er imidlertid ikke veldig fleksible, og en funksjon eller klasse kan, på en helt annen måte enn pakker, sees på som en mer fleksibel form for modularisering. En funksjon, for eksempel, kan kalles med forskjellige parametre, og en klasse kan utvides eller instansieres på forskjellige måter. Med en pakke er det vanligvis ingen lignende muligheter, og i de fleste programmeringsspråk er en pakke ikke stort mer enn en samling av klasser.

Tanken bak PT-mekanismen er å tillate mer fleksibel gjenbruk av en samling klasser enn det man får med vanlige pakker. Denne mekanismen gjør det blant annet mulig for et program å bruke en slik samling til flere forskjellige formål, til og med i samme program. Dessuten åpner mekanismen også for at en slik samling både kan utvides og modifiseres på forskjellige måter når den anvendes i et program. Dette ligner litt på hvordan klasser allerede kan utvides ved hjelp av arv, eller parametriseres.

En slik samling av klasser kaller vi altså ikke for en pakke, men nettopp en *package template*, eller, for enkelthets skyld, bare en *template*. Ordet template antyder at dette altså ikke er en vanlig pakke, og et program må eksplisitt *instansiere* en template før klassene i templaten kan tas i bruk på vanlig måte i programmet. Dette gjøres i løpet av kompileringen og ikke mens programmet kjører. Resultatet av en slik instansiering er at klassene i templaten settes inn i programmet når instansieringen behandles av kompilatoren. Det at klassene stammer fra en template vil ikke være synlig når programmet kjører. Kun klassene i templaten, som har blitt satt inn som følge av instansieringen, vil da være synlige. Etter kompilering vil programmet fortone seg som en vanlig samling av klasser, og de instansierte template-klassene kan brukes på samme måte som vanlige klasser.

Instansiering av en template skiller seg altså fra det vi ofte forbinder med instansiering, nemlig *instansiering av klasser*. Instansiering av en klasse gjøres mens programmet kjører, men instansiering av en template gjøres under kompileringen. I denne oppgaven brukes ordet *instansiering* hovedsakelig på denne foreløpig litt uvante måten, nemlig i forbindelse med instansiering av templater.

Boo[2] er et forholdsvis nytt objektorientert programmeringsspråk laget av Rodrigo B. de Oliveira. Det ble lansert i 2003, og kan neppe sies å være blant de mest kjente, men det har hele tiden vært under videre utvikling, og språket har mange interessante egenskaper. Boo er et generelt program-

meringsspråk og egner seg for alt fra webapplikasjonsutvikling til spillprogrammering. I tillegg har det mekanismer for metaprogrammering som blant annet gjør det spesielt egnet til utvikling av nye, typisk mer spesialiserte, programmeringsspråk, gjerne kalt *domenespesifikke språk* ("Domain Specific Languages" eller DSL-er). Dessuten baserer språket seg på en svært fleksibel kompilatorarkitektur.

De Oliveira beskriver i sitt Boo-manifest[2] motivasjonen for å utvikle dette språket, og noe av denne gjenspeiles også i slagordet "a wrist friendly language for the CLI". Det spiller antagelig på at Boo er et språk med kompakt og kortfattet syntaks, samt at Boo baserer seg på programvaremiljøet Common Language Infrastructure[3], som er en del av .NET-plattformen til Microsoft.

Språket Boo er ikke utviklet med støtte eller initiativ fra noe foretak eller organisasjon, og utgjør heller ikke noen offisiell del av .NET-plattformen, men ser i all hovedsak ut til å være et resultat av en enkelt persons innsats. Så det er et omfattende arbeid de Oliveira har gjort og som han stadig viderefører. Etter hvert som interessen for språket har økt, har det også blitt flere som deltar i utviklingen av språket. Boo er dessuten fri programvare, og utviklingen ser ut til å være organisert på lignende måte som annen programvare i denne kategorien. Kildekoden er tilgjengelig for alle som er interessert, og i prinsippet kan hvem som helst bidra til utviklingen av språket, ved å rapportere eller rette feil, foreslå endringer, delta i diskusjoner og så videre. Det ser imidlertid uansett ut til at det fortsatt er de Oliveira som er den som i all hovedsak driver utviklingen av språket fremover.

Hovedfokus for denne oppgaven når det gjelder Boo er altså å undersøke og vurdere de muligheter for metaprogrammering som dette språket tilbyr. Boo tilrettelegger for å lage rutiner for å modifisere eller generere deler av Boo-programmer, og tilbyr flere mekanismer for å manipulere programmer på denne måten under kompileringen. Mer spesifikt skal vi se på tre slike mekanismer som finnes i Boo: *metametoder*, *makroer* og *syntaktiske attributter*.

Videre har oppgaven vært å vurdere hvor godt disse egenskapene egner seg til å implementere PT-mekanismen. Vi skal se at dette faktisk er mulig å gjøre på en grei måte, og under arbeidet med denne oppgaven er disse mekanismene brukt til å implementere en enkel variant av PT for Boo. Jeg har valgt å kalle denne implementasjonen for *BooPT*.

Denne implementasjonen dekker bare de mest grunnleggende egenskapene til PT, og vi skal se at det har vært noen utfordringer knyttet til implementasjonen blant annet på grunn av begrensinger i Boo. Vi skal blant annet se at Boo har begrensede muligheter for å utvide syntaksen i språket. Dette har lagt begrensninger når det gjelder valg av syntaks for BooPT. Dessuten er det en utfordring knyttet til det å velge et passende lagringsformat for

kompilerte templater. Disse punktene skal vi komme nærmere inn på senere.

Innholdet i denne oppgaven er organisert som følger. Vi starter en kort beskrivelse av PT-mekanismen i kapittel 2. Så følger en introduksjon til Boo i kapittel 3 med spesiell vekt på kompilatorarkitektur og mekanismene språket har for metaprogrammering. I kapittel 4 gis en beskrivelse av BooPT, og en relativt detaljert gjennomgang av implementasjonen. Deretter kommer en vurdering av implementasjonen og hvordan metaprogrammeringsmekanismene i Boo har vært egnet til dette. Til slutt, i kapittel 5, kommer en oppsummering og det gis forslag til muligheter for videre arbeid på dette området, først og fremst med tanke på forbedring av BooPT.

Kapittel 2

Package Templates

Jeg vil nå gi en noe mer utfyllende forklaring av PT og beskrive noen viktige egenskaper ved denne mekanismen. PT-mekanismen ble først beskrevet i [1], og [4] beskriver PT i større detalj og diskuterer en del utfordringer knyttet til implementasjon av denne mekanismen. De viktigste egenskapene til PT-mekanismen listes opp i [1]:

- Omnavning
- Utvidelse
- Bevaring av klassehierarkier
- Flerbruk
- Klassesammenslåing
- Typeparametrisering

For å forklare disse egenskapene nærmere vil vi se på et eksempel som går ut på å programmere en grafstruktur, bestående av noder og kanter. Dette er et eksempel på en anvendelse av PT som er nevnt i [1]. En slik struktur er nyttig til å beskrive mange forskjellig typer fenomener, for eksempel et veinettverk bestående av veier og veikryss eller et nettverk av flyplasser og flyruter. Et konkret eksempel på hvordan en template for en slik graf kan se ut er gitt under.

```
template Graph
{
  class Node
  {
    Edge firstEdge, lastEdge;
    void insertEdge(Node to) { ... }
  }
```

```
  class Edge
  {
    Node from, to;
    Edge prevEdge, nextEdge;
    void removeMe() { ... }
  }
}
```

Vi ser altså at en template er en samling av klasser som gis et templatenavn. Jeg har her brukt Java-lignede syntaks for å beskrive PT, slik det også er gjort i den opprinnelige beskrivelsen av PT[1]. PT-mekanismen er imidlertid et generelt konsept, som ikke er knyttet til noe bestemt programmeringsspråk, slik at en implementasjon av PT i et bestemt språk naturlig vil bruke en syntaks som passer til språket. Denne oppgaven har blant annet som mål å beskrive en implementasjon av PT for språket Boo, så en slik variant av PT vil naturlig nok ha en syntaks som er mer tilpasset Boo. Dette er beskrevet mer detaljert i seksjon 4.2.

Før klassene i en template kan tas i bruk må templaten instansieres. En instansiering av en template kan gjøres i en vanlig pakke som kan utgjøre en del av et program eller et programbibliotek. Men en template kan også instansieres i definisjonen av en annen template. Når jeg heretter snakker generelt om instansiering av en template vil jeg oftest bare si at templaten instansieres i et program, og la dette dekke alle tilfeller. Den enkleste måten å instansiere `Graph`-templaten over er som følger:

```
inst Graph;
```

Dette gjør at klassene definert i templaten settes inn i programmet der denne instansieringen er angitt. Dette skjer altså i løpet av kompileringen av programmet. Disse klassene vil etter dette fremstå som helt vanlige klasser, og som når programmet kjøres kan brukes til å opprette objekter på lik linje med alle andre klasser. Klasser som stammer fra templater vil jeg kalle for *template-klasser*.

En template er altså mye mer fleksibel enn en vanlig pakke, blant annet ved at den kan instansieres flere ganger, slik at den kan brukes til forskjellige formål i samme program. Dette forutsetter, som vi skal se under, at klassene i templaten må gis nye navn i forbindelse med minst en av instansieringene.

Vi skal i følgende seksjoner se nærmere på de sentrale egenskapene ved PT, blant annet med utgangspunkt i eksemplet over.

2.1 Omnavning

Klassene i en template kan få nye navn i det templaten instansieres. Dette er nyttig av flere grunner. For det første, dersom templaten instansieres for å benyttes til et bestemt formål kan det gjøre at programmet blir lettere å forstå. For eksempel dersom vi bruker graf-templaten definert over som utgangspunkt for å modellere et nettverk av veier og veikryss, slik at klassen `Edge` brukes til å representere veier og `Node` brukes til å representere veikryss, vil det være naturlig at disse gis nye mer beskrivende navn, for eksempel henholdsvis `Road` og `Junction`.

En annen grunn er altså at en template kan instansieres flere ganger i samme program, og dette forutsetter at klassene som opprettes i den ene instansieringen må ha forskjellige navn fra klassene som opprettes i den andre. En tredje grunn er at to forskjellige templater kan ha en klasse med samme navn. Dette kan gi en utilsiktet klassesammenslåing, slik som forklart i seksjon 2.5. Dette må i så fall løses ved at minst en av klassene må gis ett nytt navn.

At det er nyttig å instansiere en template flere ganger blir tydelig når vi ser på graf-eksemplet. For samtidig som det er nyttig å bruke en slik struktur til å modellere et nettverk av veier og kryss, kan det for eksempel være ønskelig å bruke den til å modellere et nettverk av flyplasser og flyruter.

Også medlemmene i en klasse, det vil si metode- og feltdeklarasjoner, kan omnavnes. Følgende er et eksempel på hvordan vi kan instansiere graf-templaten og samtidig sørge for at både klasser og medlemmer i disse får nye navn:

```
inst Graph with Node => Junction(firstEdge -> firstRoad,
                                 lastEdge  -> lastRoad,
                                 insertEdge(Node) -> insertRoad),
                Edge => Road(prevEdge -> prevRoad,
                             nextEdge -> nextRoad);
```

2.2 Utvidelse av klasser ved instansiering (adds-klasser)

Klasser som blir opprettet når en template instansieres kan utvides med nye metode- og feltdeklarasjoner. Dette er nyttig siden en template typisk utformes med tanke på å kunne brukes til mange forskjellige formål. Template-klassene vil derfor i utgangspunktet inneholde generell funksjonalitet, som er nødvendig for at template-klassene skal kunne virke sammen og utføre sine oppgaver Når templaten tas i bruk til spesifikke anvendelser, vil det imidlertid være behov for å tilpasse de instansierte template-klassene til an-

vendelsesområdet. For eksempel kan en template for en grafstruktur ha mye nyttig og generell funksjonalitet for blant annet traversering og annen behandling av grafer. Om vi ønsker å bruke denne templaten til å modellere for eksempel et veinettverk, så er det opplagt at det er nødvendig med ekstra funksjonalitet og ekstra informasjon. Dette løses ved at man i forbindelse med instansieringen av en template angir nye metode- og feltdeklarasjoner for de instansierte template-klassene.

Her er et eksempel på hvordan vi kan utvide klassene som blir resultatet av instansieringen av `Graph`-templaten over med de gitte omnavningene.

```
class Junction adds
{
  int junctionNo;
  bool trafficLights;

  void insertRoad(Junction to)
  {
    tsuper.insertRoad(to);
    ...
  }

  void insertOnewayRoad(Junction to)
  {
    ...
  }
}

class Road adds
{
  string roadType;
  bool oneway;
  int lanes;
}
```

Vi ser at deklarasjonen av en utvidelse ligner mye på en vanlig klassedefinisjon der klassenavnet etterfølges av ordet `adds`. Dette kaller vi for `adds`-klasser, selv om det ikke egentlig er selvstendige klassedefinisjoner. Det er kun utvidelser av eksisterende template-klasser som har blitt instansiert med `inst`. Vi ser at dette åpner for muligheten for å legge til helt nye metode- og feltdeklarasjoner, men også for å redefinere ("override") metoder som allerede finnes i template-klassene. Metoden `insertRoad` er for eksempel allerede definert i `Graph`-templaten (riktignok som `insertEdge` men den gis nytt navn i forbindelse med instansieringen). Det er som vanlig kun metoder, ikke feltdeklarasjoner, som kan redefineres på denne måten. Når en metode, `insertRoad` i vårt eksempel, blir redefinert, kan det være nyttig å fortsatt ha mulighet til å kalle den opprinnelige utgaven av metoden fra metoder i `adds`-klassen. Over ser vi for eksempel hvordan vi i den nye definisjonen av `insertRoad` kan bruke ordet `tsuper` for å kalle den opprinnelige utgaven av metoden.

Denne form for utvidelse av klasser kan ligne litt på vanlig objektorientert arv fra super- til subklasser. Men den skiller seg altså på det vesentlige punktet at når en klasse utvides med `adds` så blir ikke resultatet en helt ny klasse ved siden av den eksisterende, men snarere en utvidelse av den eksisterende klassen. Dette skjer dessuten i løpet av kompileringen. Til sammenligning blir resultatet av vanlig objektorientert arv to klasser, den gamle klassen – superklassen, og den nye – subklassen, som begge er tilgjengelig, og kan brukes til å opprette objekter, når programmet kjøres.

2.3 Bevaring av klassehierarkier

En annen egenskap ved PT som nevnes i [1] er at hierarkier av super- og subklasser i en template vil bevares når templaten instansieres. Dette gjelder også når template-klassene gis nye navn eller utvides med `adds`-klasser. Anta for eksempel at templaten `T` har en klasse `A`, og en klasse `B` som er en subklasse av `A`. Anta videre at `T` instansieres på en slik måte at `A` blir utvidet med en `adds`-klasse og får navnet `C`, mens `B` får nytt navn `D`. Da vil `C` være superklassen til `D`. Klassehierarkiet mellom `A` og `B` bevares altså når de utvides eller omnavnes.

2.4 Flere instansieringer av samme template

Som nevnt over tilrettelegger template-mekanismen for flerbruk ved at en template kan instansieres flere ganger i samme program. Vi så et eksempel på hvordan templaten `Graph` ble instansiert for å modellere et nettverk av veier. Om vi i det samme programmet i tillegg ønsker å lage en modell av for eksempel et nettverk av flyplasser og flyruter kan vi bruke samme template:

```
inst Graph with Node => Airport(insertEdge(Node) -> insertFlight),
                Edge => Flight;
```

2.5 Klassesammenslåing

En viktig egenskap ved PT er at klasser fra to (eller flere) instansierte templater kan slås sammen til en klasse. Slik klassesammenslåing angis ved bruke syntaksen for omnavning. For å angi at to klasser fra forskjellige template-instansieringer skal slås sammen til en, må nemlig klassene omnavnes slik at de får samme navn. Resultatet blir da én klasse med dette navnet som består av alle metoder og felt fra hver av de opprinnelige klassene. Dersom to templater som instansieres har klasser med samme navn, vil slike klasser automatisk bli slått sammen, om de da ikke gis nye og forskjellige navn i

forbindelse med instansieringen. Det er antagelig ikke så ofte at klasser som i utgangspunktet har samme navn faktisk skal slås sammen, så det blir dermed viktig å passe på at de gis forskjellige navn slik at de ikke blir slått sammen uten at det er meningen.

For eksempel, hvis vi har to templater `T` og `U` og som har henholdsvis klassene `A` og `B`, og `T` og `U` instansieres i samme program, så kan `A` og `B` slås sammen til klassen `C` ved å gjøre de rette omnavningene:

```
inst T with A => C;
inst U with B => C;
```

Dette vil resultere i klassen `C`, som består av alle metoder og felt fra både `A` og `B`. Etter en slik sammenslåing vil klassene `A` og `B` ikke eksistere hver for seg, bare som en kombinasjon i form av klassen `C`. Dersom klassene `A` og `B` har felt med samme navn eller metoder som har samme signatur, må disse omnavnes for å unngå navnekonflikter i den sammenslåtte klassen.

En tilsvarende utfordring har vi for konstruktørene til klasser som skal sammenslås, men en konstruktør kan ikke omnavnes på samme måte som en vanlig metode. Vi krever derfor at en sammenslått klasse må definere nye konstruktører i `adds`-klassen til `C`. Konstruktørene i de opprinnelige klassene `A` og `B` kan isteden kalles fra konstruktøren i `C`, og til dette brukes en variant av `tsuper` beregnet på konstruktører. For eksempel:

```
class C adds
{
  C()
  {
    tsuper[T]();
    tsuper[U]();
  }
}
```

For å angi hvilken av de opprinnelige konstruktørene som skal kalles er det altså nødvendig å angi hvilken template den kommer fra.

PT-mekanismen angir regler for å unngå at en klasse har flere superklasser (multippel arv). Dette er mest aktuelt i forbindelse med klassesammenslåing, ettersom to klasser som blir slått sammen til en kan ha forskjellige superklasser. Kort fortalt unngås dette ved å kreve at også superklassene til sammenslåtte klasser må slås sammen. Dessuten er det tillatt ved sammenslåing av klasser at en av disse har en superklasse som ikke er definert i templaten. Da må ingen av de andre klassene i sammenslåingen ha superklasser. Dette må i så fall deklareres med det spesielle nøkkelordet `external` når en slik klasse defineres i en template.

2.6 Typeparametrisering

En template kan ha typeparametre på lignende måte som for eksempel Java-klasser:

```
template T<X>
{
  class A
  {
    X a;
    ...
  }

  class B
  {
    X b;
    ...
  }
}
```

Her ser vi at templaten `T` har `X` som typeparameter, og klassene i templaten bruker denne i noen av feltdeklarasjonene. På denne måten er det altså mulig å angi en type når templaten instansieres slik at forekomster av typeparameteren `X` blir erstattet med en ekte type når templaten instansieres. En slik typeparameter kan gis en begrensning ("bound"), for eksempel at den må være subklasse av en annen gitt klasse.

I [4] beskrives det dessuten også hvordan templater kan ha andre templater som parametre.

2.7 Template-hierarkier

Templater kan også instansieres inne i andre templater. Vi kan derfor få hierarkier av templater som bygger på hverandre med `adds`-klasser. En klasse i en template kan altså utvides med en `adds`-klasse mer enn kun en gang, ved at templaten med klassen instansieres inne i en annen template. Vi kan for eksempel ha en template `T`, og så en template `U` som instansierer `T` og utvider en av klassene i `T` ved å bruke `adds`:

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

template U
{
  inst T with A => B;
  class B adds { ... }
}
```

Templaten `U` kan til slutt instansieres i en pakke `P`:

```
inst U with B => C;
class C adds { ... }
```

Dette ligner litt på vanlig arv fra super- til subklasser, men som nevnt i seksjon 2.2 er det samtidig vesentlige forskjeller, blant annet ved at klassene `A` og `B` ikke er med i `P`, og vil således ikke være tilgjengelig når programmet kjøres. I [4] betraktes dette som at klasser i en template kan utvides langs to dimensjoner, en subklasse-dimensjon og en `adds`-dimensjon.

Kapittel 3

Boo

Boo er et imperativt og objektorientert språk med statisk typing. For å gi et inntrykk av dette språket er det vist et lite Boo-program som eksempel i figur 3.1. Programmet demonstrerer bare noen av de mest grunnleggende elementene i språket. Det inneholder en klasse med en metode som beregner primtallsfaktoriseringen til heltall større enn to, og det tar for enkelthets skyld heller ikke sikte på å være veldig optimalt.

De Oliveira lister opp en del egenskaper ved språket[2], og noen av disse avspeiles i eksempelprogrammet:

Python-lignende syntaks Syntaksen ligner som vi ser veldig på språket Python, blant annet ved at linjeskift brukes til å avslutte setninger og innrykk brukes for å styre blokkinndelingen av programmet. Vi ser dessuten at det er nødvendig å bruke et spesielt symbol dersom en setning skal deles over flere linjer: \.

Støtte for vanlige programmeringsmønstre Boo har innebygget støtte for en del vanlige programmeringsmønstre. Et enkelt eksempel er at `PrimeFactorization` definerer en metode `ToString`. Dette gjør det mulig å referere til objekter av denne klassen i sammenhenger der det er forventet en verdi av datatypen `string`, og i slike tilfeller vil metoden `ToString` automatisk kalles og det er returverdien fra denne som blir brukt. Dette er et programmeringsmønster som forenkler utskrift av tekst knyttet til objekter.

Typeinferens Dersom typen til en variabel eller returverdien til en funksjon kan bestemmes automatisk av kompilatoren, er det ikke nødvendig å eksplisitt deklarere denne. I funksjonen i eksemplet ser vi for eksempel at variabelen `factor` opplagt er av typen `List`, så det er nødvendig å deklarere denne eksplisitt (`factors as List`). Likeledes kan kompilatoren dermed fastslå at returverdien til funksjonen også er en liste.

```
import System

class PrimeFactorization:
  [property(IntNumber)]
  _intNumber as int

  def constructor(n as int):
    _intNumber = n

  def Factorize():
    return Factorize(_intNumber)

  static def Factorize(n as int):
    factors = List()
    generator = j for j in range(2, Math.Sqrt(n) + 1) \
      if j == 2 or j % 2 == 1
    for i in generator:
      while n % i == 0:
        n /= i
        factors.Add(i)
    if n != 1:
      factors.Add(n)
    return factors

  def ToString():
    return join(Factorize(_intNumber))

try:
  n = int.Parse(argv[0])
except e as Exception:
  print "invalid input:", n
  Environment.Exit(1)

print "$n:", PrimeFactorization(n)
```

Figur 3.1: Boo-program som beregner primtallsfaktorisering

Implisitt deklarasjon av variabler Typen til variabler kan deklareres med nøkkelordet `as`. For lokale variabler er det imidlertid ikke nødvendig å deklarere typen til en variabel dersom denne kan bestemmes automatisk av kompilatoren, men det er påkrevd at det angis en tilordning av variabelen før den brukes i andre uttrykk. Merk at dette kravet gjelder kun den tekstlige forekomsten av tilordningen i forhold til senere bruksforekomster. Det er ikke et krav at tilordningen faktisk skal utføres før variabelen brukes i et uttrykk.

Dette ser vi et eksempel på med variabelen `n` i faktoriseringsprogrammet. Det er ikke sikkert at tilordningen av `n` i `try`-setningen lykkes slik at den første faktiske bruken av variabelen kan være i `except`-delen og verdien som da skrives ut er standardverdien `0`.

Den første tekstlige forekomsten av en tilordning av variabelen kan derfor regnes som en slags deklarasjon selv om det som sagt ikke er et krav om at den utføres. Denne måten å deklarere variabler på kan kanskje oppfattes som noe forvirrende.

For øvrig, kan det i denne sammenhengen også være verdt å merke seg at nøsting av skop ikke forekommer i Boo. I en gitt funksjon er det kun ett skop, der alle lokale variabler er synlige. Eller for nyansere noe, så kan en si at en lokal variabel er synlig fra der den deklareres, enten det er eksplisitt eller implisitt, og ut skopet til funksjonen. Dette gjelder altså også om variabelen er deklarert inne i blokken til for eksempel en løkke.

Klasser og funksjoner ikke påkrevd Korte Boo-programmer kan skrives uten at det er nødvendig å legge de inn i en egen funksjon eller klasse, for eksempel i motsetning til C# eller C som henholdsvis enten krever en hovedklasse eller -funksjon som inneholder oppstartskoden for programmet.

Høyere-ordens funksjoner Funksjoner kan på samme måte som verdier av vanlige datatyper slik som `int` og `string` lagres i variabler, returneres fra eller gis som parametre til funksjoner.

Generatoruttrykk og -funksjoner Et generatorutrykk er en sekvens uttrykt ved hjelp av en `for`-løkke. Dette kan være en potensielt uendelig sekvens, og elementer eller deler av sekvensen kan hentes ut. Generatorer kan i likhet med funksjoner lagres i variabler, returneres fra eller gis som parametre til funksjoner. Eksempelprogrammet viser en generator som lager en sekvens av tallet 2 og alle etterfølgende oddetall opptil et gitt tall. En generatorfunksjon ligner på en vanlig funksjon, men genererer på samme måte som et generatoruttrykk en (mulig uendelig) sekvens av elementer.

Mulighet for dynamisk typing Boo er i utgangspunktet et statisk typet språk, men åpner likevel for at enkelte variabler kan ha dynamisk typing. Det vil si at typen til en variabel bestemmes og sjekkes under kjøringen av programmet. Dette kalles i Boo for *duck typing*, og slike variabler deklareres med `duck` som type.

Metametoder, makroer og syntaktiske attributter Dette er metaprogrammeringsmekanismer for å modifisere et program under kompileringen. Dette er svært kraftige mekanismer som det fokuseres mye på i denne oppgaven. Vi ser et eksempel på bruk av et syntaktisk attributt i eksempelprogrammet i figur 3.1. Attributtet `property` er gitt i hakeparenteser for feltet `_intNumber` i klassen `PrimeFactorization`. Dette oppretter automatisk en *egenskap* `IntNumber` som knyttes til feltet `_intNumber` i denne klassen:

```
IntNumber as int:
  get:
    return _intNumber
  set:
    _intNumber = value
```

En egenskap ("property") er en tredje type klassemedlem, i tillegg til metoder og felt. En egenskap kan leses fra og tilordnes verdier som om det var en variabel, men det som egentlig skjer er at en `get`- eller `set`-metode kalles. Attributtet `property` er et predefinert syntaktisk attributt, men senere skal vi se på hvordan man selv kan definere slike syntaktiske attributter, samt metametoder og makroer.

Utvidbar kompilatorarkitektur Boo tilrettelegger for at man enkelt skal kunne utvide kompilatoren eller påvirke kompileringen også på mer drastiske måter. Dette blir også til en viss grad utnyttet i denne oppgaven.

Når det gjelder programmeringsspråk er det oftest viktig å skille mellom språk og implementasjon, siden det ofte kan være forskjeller mellom implementasjoner, og det kan dermed være forskjellige varianter av et språk. Boo er som nevnt et ganske nytt språk og det finnes for tiden kun én implementasjon. Skillet mellom språk og implementasjon blir dermed mindre viktig, og derfor bruker jeg også Boo som betegnelse på begge. Det vil som regel være mindre relevant eller være opplagt ut i fra sammenhengen hvorvidt det er snakk om språk eller implementasjon. Faktisk, når det gjelder Boo så finnes det ingen formell spesifikasjon av språket og språket kan vel sies å være definert ut i fra implementasjonen til de Oliveira.

Dessuten er det verdt å merke seg at metaprogrammeringsmekanismene i Boo er tett knyttet opp mot implementasjonen siden bruk av disse innebærer at man kan påvirke kompilatoren på forskjellig vis, blant annet ved at man har tilgang interne datastrukturer i kompilatoren.

Med slike muligheter for metaprogrammering er det også mulig å utvide språket og lage varianter, for eksempel DSL-er. Men slike varianter vil uansett ha det til felles at de bygger på den samme grunnimplementasjonen av Boo.

Det finnes for øvrig også lite annen dokumentasjon som tar sikte på å dekke hele språket. Dette kan være en utfordring når en skal lære seg Boo, og ikke minst om en ønsker å sette seg inn i de mer avanserte mulighetene som er utnyttet i denne oppgaven. Det er samlet en del materiale på websiden til Boo[5], blant annet *Boo Primer*[6], men i likhet med mye av det andre som er tilgjengelig, er dette en innføring som kun dekker de grunnleggende elementene i språket. Det finnes en *Language Guide*[7] som også dekker mer avanserte emner, blant annet deler av kompilatorarkitekturen, men den er

relativt ufullstendig. Det er verdt å merke seg at språket fortsatt er under aktiv utvikling, og det er med jevne mellomrom gjenstand for både større og mindre forandringer, og av og til introduseres helt nye egenskaper. Slike endringer dokumenteres imidlertid ofte nokså sporadisk og uformelt, sånn som i blogginnlegg, epostlister og lignede.

Det er ennå heller ikke utgitt mye trykt litteratur om Boo. Ayende Rahien har imidlertid skrevet en bok *DSLs in BOO*[8], som både gir en kort innføring i de elementære delene av språket, samt at den omhandler de mer avanserte mulighetene for metaprogrammering i Boo. Likevel, slik tittelen indikerer, så dreierer boken seg først og fremst om utvikling av domenespesifikke språk eller DSL-er. Rahien beskriver hvordan Boo egner seg svært godt til dette formålet, og har derfor valgt å basere seg på Boo i en bok som først og fremst handler om DSL-er.

Når det kommer til stykket er det kanskje selve kildekoden[9] som er den beste kilden til å lære om de avanserte egenskapene til Boo. Språket er som nevnt i kontinuerlig utvikling, og kildekoden må dermed betraktes som den eneste oppdaterte og nøyaktige dokumentasjonen som finnes. Det er sparsomt med kommentarer til koden, men den er oversiktlig og forholdsvis lett forståelig. Med koden følger det også noen eksempler som det kan være nyttig å studere, og dessuten enhetstester som tar sikte på å dekke hele språket og de kan dermed gi utvidet innsikt.

Det er også noen flere hjelpemidler tilgjengelig på Internett som gjør det lettere å sette seg inn i språket og den tilhørende implementasjonen. Det finnes et syntaksdiagram for språket[10], en oversikt over klasser i det abstrakte syntakstreet[11] og en mer omfattende oversikt over hele API-et[12] til klassehierarkiet som brukes i Boo-kompilatoren.

Gjeldende versjon av Boo er 0.9.4[13], og denne er implementert i C#. I følge websiden til Boo vil versjon 1.0 av språket bli utgitt i det Boo selv er tatt i bruk som implementasjonsspråk.

Til tross for det noe beskjedne versjonsnummeret og det faktum at Boo er under kontinuerlig utvikling, så er det verdt å påpeke at mange viktige og grunnleggende egenskaper til språket er både velprøvde og stabile. For eksempel Rahien skriver i sin bok[8] blant annet om hvordan Boo egner seg til bruk i produksjonssystemer. Slik sett kan Boo betraktes som et modent språk, selv om mer avanserte egenskaper, for eksempel for metaprogrammering og utvidbar syntaks, er under stadig utvikling.

3.1 Kompilering og kjøring av Boo-programmer

Boo er som nevnt basert på .NET-plattformen og programvaremiljøet Common Language Infrastructure[3] som er et velutviklet rammeverk for kom-

pilering og kjøring av programmer. Dette tilbyr blant annet et rikt klassebibliotek, det såkalte Base Class Library (BCL), som programmer kan benytte seg av. Dessuten brukes en standard .NET-komponent som kalles Common Language Runtime (CLR) for kjøring av programmene.

Boo baserer seg i likhet med andre språk med utgangspunkt i .NET-plattformen altså på kompilering av koden. Koden kompileres ikke direkte til maskinkode men til Common Intermediate Lanugage (CIL) som er en bytekode-variant som brukes i .NET. Resultatet av en Boo-kompilering er altså et .NET-assembly, enten i form av en DLL- eller EXE-fil avhengig av om det er et programbibliotek eller et direkte kjørbart program som lages.

Når programmet skal kjøres, så gjøres dette ved hjelp CLR-komponenten i .NET. Denne oversetter bytekoden til maskinkode som kan kjøres direkte på maskinen. Det finnes implementasjoner av CLR for en god del plattformer. Av de viktigste er den offisielle .NET-implementasjonen til Microsoft som støtter Windows-operativsystemer, og Mono som blant annet støtter UNIX og forskjellige mobile plattformer. Slik kan altså Boo-programmer kjøres på en rekke forskjellige maskinvarearkitekturer.

Filer med Boo-kildekode gis typisk et navn med endelsen `.boo`, og hvis vi antar at faktoriseringsprogrammet over er lagret i filen `factor.boo`, kan vi kompilere dette ved å gi en enkel kommando: `booc factor.boo`. Dette vil gi en EXE-fil med kjørbar kode. Dersom programmet er avhengig av ekstra programbiblioteker i form av andre .NET-assemblier må disse kobles inn under kompileringen ved å bruke opsjonen `-r`. Hvis for eksempel faktoriseringsprogrammet hadde krevd et slikt assembly, si `MathLib.dll`, måtte vi kompilere det med følgende kommando: `booc -r:MatLib.dll factor.boo`.

3.2 Kompilatorarkitektur

Boo har en veldig fleksibel kompilatorarkitektur, som på mange måter kan sies å være en viktig del av språket selv. Boo tilbyr et omfattende grensesnitt mot kompilatoren som en del av sitt standard klassebibliotek. Disse er tilgjengelig under navnerommet `Boo.Lang`, og spesielt `Boo.Lang.Compiler` inneholder mange sentrale klasser.

Grensesnittet mot kompilatoren er også tilgjengelig for kjørende Boo-programmer, først og fremst ved at de enkelt kan iverksette en egen instans av Boo-kompilatoren. Oppstart av kompilatoren gjøres ved hjelp av klassen `Boo.Lang.Compiler.BooCompiler`. Et kort program, for først å kompilere og deretter kjøre et annet Boo-program, lagret i filen `script.boo`, kan se slik ut:

```
import Boo.Lang.Compiler

compiler as BooCompiler = BooCompiler()
compiler.Parameters.Pipeline = Run()
compiler.Parameters.Input.Add(FileInput("script.boo"))
context as CompilerContext = compiler.Run()
```

Dette er kanskje noe man i første omgang ikke forestiller som så veldig nyttig med mindre det man lager er selve Boo-kompilatoren. Men dette er en teknikk som kan være nyttig i systemer som for eksempel krever høy grad av tilpasning eller avansert konfigurasjon. En måte å løse slike krav på er lage et eget programmeringsspråk spesielt beregnet på en bestemt applikasjon. Dette kalles som nevnt gjerne for DSL-er, og Boo er spesielt godt egnet til å lage kompilatorer for slike språk.

Denne funksjonaliteten er dessuten heller ikke bare forbeholdt Boo-programmer. Siden Boo-klassebiblioteket er basert på .NET, så kan det brukes i alle språk for denne plattformen. Dermed kan man for eksempel kalle Boo-kompilatoren fra C#-program om man ønsker det, slik at man for eksempel kan benytte seg av DSL-er basert på Boo i applikasjoner som ikke selv er skrevet i Boo.

3.2.1 Kompilatorsekvensen

Den sentrale delen i Boo-kompilatoren er en sekvens av såkalte kompilatorsteg ("steps"). Gjennom disse stegene transformeres et Boo-program fra kildekode til ferdig kompilert .NET-assembly. Det første steget er parseren som oversetter programmet til et abstrakt syntakstre (AST). Dette er en tre-representasjon av den syntaktiske oppbygningen til programmet, som vi skal se nærmere på i seksjon 3.2.2

Hvert steg gjør typisk en omforming av, eller på annen måte behandler, det abstrakte syntakstreet til programmet. Under denne prosessen kan AST-et betraktes som et resultat fra ett steg som gis videre til neste steg i sekvensen. Sekvensen av disse stegene kalles i Boo derfor ganske betegnende for en *pipeline*. Jeg vil imidlertid bruke betegnelsen kompilatorsekvens på norsk.

Hele kompilatorsekvensen til Boo er konfigurerbar, og kan endres eller byttes ut i sin helhet, dersom man er interessert i å endre på oppførselen til kompilatoren. I prinsippet ville det faktisk være mulig å lage en helt ny kompilatorsekvens, for eksempel med en parser for et helt annet språk enn Boo, og en kodegenerator som genererer noe ganske annet enn .NET-kode, for eksempel maskinkode. Likevel, vil nok de fleste i praksis basere seg på standard-sekvensen i Boo og de kompilatorsteg som allerede er tilgjengelig.

Kompilatorsekvensen består som sagt av en rekke steg, i siste versjon faktisk hele 57 steg. Disse er listet opp kolonnevis under.

```
Parsing                              ReifyTypes
PreErrorChecking                     TypeInference
MergePartialTypes                    InjectImplicitBooleanConversions
InitializeNameResolutionService      ConstantFolding
IntroduceGlobalNamespaces            CheckLiteralValues
TransformCallableDefinitions         OptimizeIterationStatements
BindTypeDefinitions                  BranchChecking
BindGenericParameters                VerifyExtensionMethods
ResolveImports                       CheckIdentifiers
BindBaseTypes                        StricterErrorChecking
MacroAndAttributeExpansion           DetectNotImplementedFeatureUsage
MergePartialTypes                    CheckAttributesUsage
ExpandAstLiterals                    ExpandDuckTypedExpressions
IntroduceModuleClasses               ProcessAssignmentsToValueTypeMembers
NormalizeStatementModifiers          ExpandPropertiesAndEvents
NormalizeTypeAndMemberDefinitions    CheckMembersProtectionLevel
NormalizeExpressions                 NormalizeIterationStatements
BindTypeDefinitions                  ProcessSharedLocals
BindGenericParameters                ProcessClosures
BindEnumMembers                      ProcessGenerators
BindBaseTypes                        ExpandVarArgsMethodInvocations
CheckMemberTypes                     InjectCallableConversions
BindMethods                          ImplementICallableOnCallableDefinitions
ResolveTypeReferences                RemoveDeadCode
BindTypeMembers                      CheckNeverUsedMembers
CheckGenericConstraints              CacheRegularExpressionsInStaticFields
ProcessInheritedAbstractMembers      EmitAssembly
CheckMemberNames                     SaveAssembly
ProcessMethodBodiesWithDuckTyping
```

Ikke alle stegene er like sentrale. Noen av de viktigste, spesielt i forbindelse med arbeidet i denne oppgaven, er følgende:

`Parsing` Dette steget gjør både leksikalsk og syntaktisk analyse og bygger syntakstreet som svarer til kildekoden. Dette syntakstreet blir videre omformet på forskjellige måter av de etterfølgende stegene i sekvensen.

`MacroAndAttributeExpansion` Dette steget behandler makroer og syntaktiske attributter. Dette er mekanismer for metaprogrammering som er sentrale i denne oppgaven, og de blir beskrevet mer detaljert i seksjon 3.3. Disse lar programmereren gripe inn i kompilatorsekvensen for å modifisere AST-et til programmet som kompileres. Når man utformer makroer eller syntaktiske attributter er det ofte nyttig å være klar over hvor i kompilatorsekvensen disse utføres, blant annet med tanke på hvilke begrensninger dette medfører. For eksempel kommer dette steget før mye av den semantiske analysen blir gjort. Makroer og syntaktiske attributter har for eksempel derfor ikke tilgang til navne- og typebindinger siden disse først opprettes i et senere steg.

`ProcessMethodBodiesWithDuckTyping` Dette steget har et mystisk navn, men utfører viktige oppgaver som er av en viss betydning for arbeidet med BooPT. For det første, setter det opp navn- og typebindinger i AST-et. Blant annet blir bruksforekomster av metode- og variabelnavn bundet til de tilsvarende deklarasjoner i AST-et. For det andre, utføres metametoder som er den tredje typen mekanisme for metaprogrammering i Boo.

`EmitAssembly` Dette steget genererer .NET bytekode som svarer til syntakstreet som er resultatet av de transformasjoner som har blitt gjort i tidligere steg.

Kompilatorsekvensen er som sagt konfigurerbar, og man kan bygge en sekvens fra bunnen ved hjelp av en blanding av eksisterende og egendefinerte steg, eller man kan ta utgangspunkt i en eksisterende sekvens og sette inn nye eller fjerne steg. For eksempel dersom man kun er interessert i å sjekke at et program er korrekt og ikke er interessert i annen output enn feilmeldinger, kan man ta utgangspunkt i den vanlige kompilatorsekvensen men for eksempel fjerne stegene som utgjør kodegenereringsdelen, nemlig `EmitAssembly` og `SaveAssembly`.

3.2.2 Det abstrakte syntakstreet

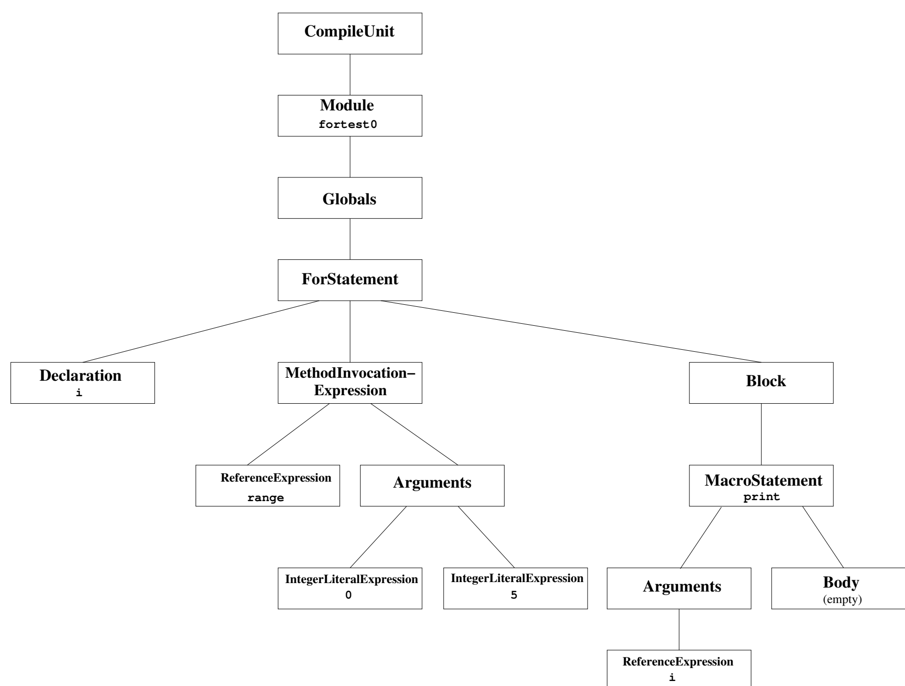
Et gitt Boo-program har et veldefinert abstrakt syntakstre (AST). Dette er som nevnt en tre-representasjon av den syntaktiske oppbygningen til programmet, og blir definert ut fra i grammatikken (syntaksen) til språket og et klassehierarki som beskriver de forskjellige typene noder som ut i fra grammatikken naturlig representerer de forskjellige konstruksjonene i språket. Syntaksen til Boo er definert med en BNF-grammatikk spesifisert for ANTLR[14], mens klassene for de forskjellige nodene i syntakstreet er definert under navnerommet `Boo.Lang.Compiler.Ast`.

Det abstrakte syntakstreet til et program er resultatet av det første kompilatorsteget (parsingen). Dette blir behandlet videre av senere steg i kompileringen. Som et eksempel på hvordan AST-et til et program kan se ut tar vi utgangspunkt i en enkel `for`-løkke:

```
for i in range(0, 5):
  print i
```

De viktigste nodene i AST-et til programmet med denne `for`-løkka er vist i figur 3.2. Vi ser her noen av de typer noder som kan forekomme i syntakstreet, slik som `ForStatement`, `Declaration`, `Block`, `ReferenceExpression` og `MethodInvocationExpression`. Disse svarer grovt sett til en `for`-setning, en deklarasjon, en blokk (sekvens av setninger), referanser (i dette tilfellet til

henholdsvis et metode- og variabelnavn) og et metodekall. Det er også verdt å merke seg at rot-noden i syntakstreet har klassen `CompileUnit`. Dessuten har alle noder i AST-et klassen `Node` som superklasse, og den inneholder en del nyttig funksjonalitet felles for alle klassene. Blant annet har alle noder egenskapen `ParenNode` som gir tilgang til foreldrenoden i syntakstreet.



Figur 3.2: Nodene i AST-et til et program med en `for`-løkke

Boo har innebygget funksjonalitet for utskrift av tekstlig representasjon av syntakstreet. Denne er tilgjengelig fra de alternative kompilatorsekvensene `ParseAndPrintAst` og `ParseAndPrintXml`, som henholdsvis benytter stegene `PrintAst` og `SerializeToXml` for å skrive ut syntakstreet. Det første genererer en forholdsvis kompakt innfiks representasjon av treet, men der en del av nodene i treet er utelatt. Det andre genererer en mer utførlig XML-representasjon som beskriver alle nodene i treet. Dette er nyttige hjelpermidler for å forstå hvordan et AST er bygget opp, og hvordan det transformeres av kompilatoren.

```
MethodInvocationExpression(
        Target: ReferenceExpression('range'),
        Arguments: (, ))ReferenceExpression('i')
```

Figur 3.3: Innfiks representasjon av AST-et til en `for`-løkke

Figur 3.3 og 3.4 viser de to utskriftstypene for AST-et til `for`-løkka over.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<CompileUnit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=
    "http://www.w3.org/2001/XMLSchema">
  <Modules>
    <Module Name="fortest0">
      <Attributes />
      <Visibility>None</Visibility>
      <Members />
      <BaseTypes />
      <GenericParameters />
      <Imports />
      <Globals>
        <Statements>
          <Statement xsi:type="ForStatement">
            <Declarations>
              <Declaration Name="i" />
            </Declarations>
            <Iterator xsi:type="MethodInvocationExpression">
              <Target xsi:type="ReferenceExpression" Name="range" />
              <Arguments>
                <Expression xsi:type="IntegerLiteralExpression">
                  <Value>0</Value>
                  <IsLong>false</IsLong>
                </Expression>
                <Expression xsi:type="IntegerLiteralExpression">
                  <Value>5</Value>
                  <IsLong>false</IsLong>
                </Expression>
              </Arguments>
              <NamedArguments />
            </Iterator>
            <Block>
              <Statements>
                <Statement xsi:type="MacroStatement" Name="print">
                  <Arguments>
                    <Expression xsi:type="ReferenceExpression" Name="i" />
                  </Arguments>
                  <Body>
                    <Statements />
                  </Body>
                </Statement>
              </Statements>
            </Block>
          </Statement>
        </Statements>
      </Globals>
      <AssemblyAttributes />
    </Module>
  </Modules>
</CompileUnit>
```

Figur 3.4: XML-representasjon av AST-et til en for-løkke

Som vi ser er XML-representasjonen mye mer omfattende, og er antagelig den som gir best forståelse av hvordan treet er bygget opp.

Det er også mulig å få kompilatoren til å skrive ut den tekstlige koden som svarerer til AST-et, eller en del av det. Alle noder i AST-et har en metode `ToCodeString` som skriver ut Boo-kode tilsvarende subtreet av AST-et som har den gitte noden som rot. Kompilatorsekvensen `CompileToBoo` benytter dette i slutten av sekvensen til å skrive ut et Boo-program som svarer til AST-et, istedet for å generere kjørbar kode.

Dette gir et Boo-program som semantisk sett skal være det samme som det opprinnelige programmet, men legg merke til at det syntaktisk mest trolig vil være forskjellig. Figur 3.5 viser utskriften fra denne kompilatorsekvensen for `for`-løkkeeksemplet.

```
public final transient class Fortest0Module(object):

        private static def Main(argv as (string)) as void:
                $1 = 0
                while $1 < 5:
                        i = $1
                        $1 = ($1 + 1)
                        System.Console.WriteLine(i)

        private def constructor():
                super()
```

Figur 3.5: Kode generert fra et AST med utgangspunkt i en `for`-løkke

Dette gir et lite innblikk i hva stegene i kompilatorsekvensen har gjort med programmet. Programmet ser ganske annerledes ut enn det opprinnelige. Blant annet har `for`-løkka blitt oversatt til en `while`-løkke og den har blitt innlemmet i en metode, i en klasse. For, som nevnt, blir AST-et til et program behandlet av stegene i kompilatorsekvensen. Noen steg transformerer AST-et, for eksempel for å optimalisere eller forenkle kode-generering, slik at AST-et mot slutten av kompileringen kan se ganske annerledes ut enn det som kom ut av parseren. Andre steg rører ikke treet, men bruker det for eksempel i forbindelse med feilsjekking eller typebinding.

Programmet i figur 3.5 er med ett unntak i seg selv et gyldig Boo-program, som på nytt kan gis til Boo-kompilatoren, kompileres og gi samme resultat som det opprinnelige programmet. Unntaket er at programmet har en variabel kalt `$1`. Syntaksen til Boo tillater imidlertid ikke dollartegn i identifikatorer, så parseren til Boo vil ikke akseptere dette programmet. Likevel kan senere steg i kompilatoren innføre navn som inneholder dollartegn. Steg som omformer AST-et på ulike måter kan ha behov for å legge inn nye variabler for eksempel, slik vi ser det har blitt gjort i figur 3.5. Variabelen `$1`

forekommer ikke i det opprinnelige programmet.

For å unngå konflikter med eksisterende navn i programmet, velger derfor Boo for enkelthets skyld å bruke et dollartegn i nye navn som innføres i løpet av kompileringen. Slike identifikatorer kan også opprettes av makroer eller syntaktiske attributter som defineres av brukeren.

Om vi av en eller annen grunn skulle ønske å kompilere koden i figur 3.5 på nytt i Boo-kompilatoren, helt fra begynnelsen av, må vi altså først sørge for at navn med dollartegn erstattes med gyldige identifikatorer. Dette er riktignok en ganske spesiell problemstilling, men senere skal vi se at dette faktisk er noe som kommer opp i forbindelse med implementasjonen av BooPT.

3.2.3 Predefinerte kompilatorsekvenser

Boo tilbyr, i tillegg til den som brukes som standard, også en del andre kompilatorsekvenser for forskjellige formål. Disse finnes i navnerommet `Boo.Lang.Compiler.Pipelines`. Vi har allerede sett noen eksempler på slike som kan brukes til utskrift av AST-et. Tabellen under gir en noe mer komplett oversikt.

Kompilatorsekvens	**Kortform**	**Forklaring**
`CompileToFile`		Kompilerer til fil (standard)
`Compile`	`-p:compile`	Kompilerer uten å generere kode
`Run`	`-p:run`	Kompilerer og kjører programmet
`ResolveExpressions`		Kjører alle steg f.o.m. parsing t.o.m. navn- og typebinding
`CompileToBoo`	`-p:boo`	Skriver ut Boo-koden som svarer til AST-et etter kompilering
`ParseAndPrintAst`	`-p:ast`	Skriver ut en innfiks representasjon av AST-et
`ParseAndPrintXml`	`-p:xml`	Skriver ut en XML-representasjon av AST-et

En del av disse har vært nyttige i denne oppgaven. For å starte Boo-kompilatoren med en alternativ kompilatorsekvens må `-p`-opsjonen benyttes. For eksempel kan man gi følgende kommando for å kompilere et program uten å generere kode:

```
booc -p:compile program.boo
```

3.2.4 Egendefinerte kompilatorsekvenser

Det er relativt enkelt å opprette egendefinerte kompilatorsteg og -sekvenser. Et kompilatorsteg defineres med en subklasse av `AbstractTransformerCompilerStep` og må implementere metoden `Run`, som må inneholde koden som skal

utføres i dette steget. Alle kompilatorsekvenser er subklasser av `CompilerPipeline` og har derfor metoder som `Add`, `Insert` og `Remove`. Man kan definere sin egen kompilatorsekvens ved å opprette et objekt av en av de predefinerte kompilatorsekvensene, og bruke disse metodene for å fjerne eller sette inn nye steg. Alternativt kan man definere sekvensen med en ny klasse som er subklasse av `CompilerPipeline` eller en av de predefinerte sekvensene. Her er et eksempel på hvordan man kan definere en ny kompilatorsekvens `CompileWithIntermediateSteps` som blant annet også bruker et egendefinert steg:

```
class PrintNextCompilerStep(AbstractTransformerCompilerStep):
  _stepNo as int
  _step as ICompilerStep
  def constructor(stepNo as int, step as ICompilerStep):
    _stepNo = stepNo
    _step = step

  override def Run():
    print "\n\n" + "step: " + _stepNo + ": " + _step


class CompileWithIntermediateSteps(CompilerPipeline):
  def constructor(step as ICompilerStep, pipeline as CompilerPipeline):
    super()
    steps = pipeline.Count
    for i in range(0, steps):
      Add(PrintNextCompilerStep(i, pipeline[i]))
      Add(pipeline[i])
      Add(step)
```

Vi ser at virkemåten til kompilatorsteget `PrintNextCompilerStep` er svært enkel. Det skriver kun ut en tekst. Slik dette steget brukes av `CompileWithIntermediateSteps` vil teksten som skrives ut være navnet på det følgende kompilatorsteget i sekvensen. Vanligvis vil et kompilatorsteg være mer avansert enn dette, og typisk gjøre transformasjoner av eller på annen måte behandle AST-et til programmet som kompileres. Kompilatorstegene har tilgang til AST-et gjennom egenskapen `CompilerContext.Current.CompileUnit` i den gjeldende kompilatorkonteksten.

Kompilatorsekvensen `CompileWithIntermediateSteps` er et eksempel på hvordan den vanlige kompilatorsekvensen til Boo kan endres slik at vi kan skrive ut detaljert informasjon mellom hvert steg, noe som kan belyse hvordan hva de forskjellige stegene gjør med AST-et. Dette er noe som har vært nyttig i arbeidet med denne oppgaven.

Et objekt av `CompileWithIntermediateSteps` opprettes ved å angi en annen kompilatorsekvens, for eksempel `Compile`, og et steg som skal settes inn mellom alle stegene i den gitte sekvensen. Hvis vi for eksempel angir steget `SerializeToXml`, vil vi få en kompilatorsekvens som skriver ut en XML-representasjon av AST-et mellom hvert kompilatorsteg:

```
class CompileAndPrintXmlForEachStep(CompileWithIntermediateSteps):
  def constructor():
    super(SerializeToXml(), CompileToMemory())
```

For å starte Boo-kompilatoren med en egendefinert kompilatorsekvens må -p-opsjonen brukes på følgende måte:

```
booc -p:CompileAndPrintXmlForEachStep,CompileWithIntermediateSteps program.boo
```

Den første komponenten angitt med `-p` er navnet på klassen til kompilatorsekvensen, mens den andre er navnet på .NET-assembliet som inneholder denne klassen.

3.3 Metaprogrammering

Boo har altså gode muligheter for såkalt metaprogrammering, det vil si at man kan bruke Boo til å generere nye deler av eller modifisere eksisterende Boo-programmer. Boo har, som vi allerede har vært inne på, tre slike mekanismer for metaprogrammering:

- Metametoder
- Makroer
- Syntaktiske attributter

Disse mekanismene gir programmereren mulighet til å bearbeide deler av AST-et til et program i løpet av kompileringen. Dette kan, litt forenklet, betraktes som rutiner som defineres av programmereren, og programmet som kompileres kan inneholde referanser til disse. Disse vil imidlertid ikke iverksettes under den vanlige kjøring av programmet, men isteden i løpet av kompileringen, og vil da kunne brukes til manipulere AST-et på forskjellig vis.

Hver av disse tre formene for metaprogrammering innebærer manipulasjon av det abstrakte syntakstreet for å modifisere et program. Vi har allerede nevnt en fjerde mulighet for å modifisere AST-et til et program, nemlig ved å definere egne kompilatorsteg og -sekvenser. Dette blir vel imidlertid ikke å regne som en form for metaprogrammering, men snarere som en måte for å utvide eller modifisere selve Boo-kompilatoren.

Det er nødvendig at metametoder, makroer og syntaktiske attributter defineres separat i forhold til hvor de blir benyttet. Et program som bruker en metametode må altså ikke inneholde selve definisjonen av metametoden. (Dette var imidlertid tillatt i tidlige utgaver av Boo.) Dette innebærer

typisk at kildekoden med disse definisjonene legges i en egen fil og kompileres til et eget .NET-assembly. Dette kan så kobles inn ved kompileringen av programmer som bruker disse mekanismene. Disse mekanismene brukes bare under kompileringen, slik at de vil ikke utgjøre noen naturlig del av ferdigkompilerte programmer.

Hvilken mekanisme som egner seg best i en gitt situasjon avhenger både av hva slags kallsyntaks som er mest passende og hvilke deler av AST-et som skal transformeres. Dette kommer vi mer konkret tilbake til når vi nå går gjennom hver av disse mer detaljert.

3.3.1 Metametoder

Både deklarasjon av og kall til metametoder ligner veldig på vanlige metodedeklarasjoner og -kall. Den sentrale forskjellen er altså at kall til metametoder ikke utføres mens programmet kjøres, men når det kompileres. Dette gjøres som nevnt i seksjon 3.2.1 i kompilatorsteget `ProcessMethodBodiesWithDuckTyping`.

Figur 3.6 viser et eksempel på en metametode. Metoden gjør en triviell forenkling av et aritmetisk uttrykk ved å fjerne forekomster av multiplikasjon med 1 og addisjon med 0 fra uttrykket. Dette er faktisk en optimalisering som Boo-kompilatoren selv ikke gjør, selv om det antagelig gjøres av JIT-kompilatoren før programmet kjøres. For eksemplets skyld har jeg her altså brukt en metametode, men det er klart at en slik optimalisering egentlig burde implementeres i et eget steg i kompilatoren.

Vi ser at en metametode deklareres ved å angi et såkalt attributt, `meta` i hakeparenteser. Et attributt er en slags merkelapp som kan knyttes til enkelte konstruksjoner i programmet. Vi skal behandle attributter mer detaljert i seksjon 3.3.3.

For øvrig ser dette ut som en vanlig metodedefinisjon, men parametrene som blir gitt til en metametode er subtrær av AST-et. Derfor må de formelle parametrene til en metametode ha typer som er subklasser av `Node`. Når en metametode kalles under kompileringen er det altså referanser til nodene i syntakstreet som svarer til de aktuelle parametrene som blir gitt til metoden. Selve metametodekallet vil i AST-et bli erstattet med returverdien til metoden etter at den er utført. Det betyr at returverdien til metoden også må være et subtre som kan settes inn i syntakstreet. Eksempelvis vil kallet `SimplifyArithIdentity(a + 1 * (b + 0))` til metametoden i figur 3.6 erstattes i AST-et med et subtre som svarer til uttrykket `a + b`.

Metametoden kan bare arbeide på de delene av AST-et som er tilgjengelig gjennom parametrene. Den har ikke tilgang til andre deler av AST-et, for eksempel ved bruk av `ParentNode`.

```
import Boo.Lang.Compiler.Ast

[meta]
def SimplifyArithIdentity(expr as Expression):
  exprStmt = ExpressionStatement(expr)
  exprStmt.Accept(SimplifyArithIdentityVisitor())
  return exprStmt.Expression

class SimplifyArithIdentityVisitor(DepthFirstTransformer):
  def OnBinaryExpression(node as BinaryExpression):
    super.OnBinaryExpression(node)
    if node.Operator == BinaryOperatorType.Addition:
      if node.Left isa IntegerLiteralExpression \
        and (node.Left as IntegerLiteralExpression).Value == 0:
        node.Right["optimized"] = true
        ReplaceCurrentNode(node.Right)
      elif node.Right isa IntegerLiteralExpression \
        and (node.Right as IntegerLiteralExpression).Value == 0:
        node.Left["optimized"] = true
        ReplaceCurrentNode(node.Left)
    elif node.Operator == BinaryOperatorType.Multiply:
      if node.Left isa IntegerLiteralExpression \
        and (node.Left as IntegerLiteralExpression).Value == 1:
        node.Right["optimized"] = true
        ReplaceCurrentNode(node.Right)
      elif node.Right isa IntegerLiteralExpression \
        and (node.Right as IntegerLiteralExpression).Value == 1:
        node.Left["optimized"] = true
        ReplaceCurrentNode(node.Left)
```

Figur 3.6: Metametode som forenkler et aritmetisk uttrykk

For øvrig er det verdt å bemerke at kall til metametoder kun kan forekomme der også vanlige metodekall er tillatt. Det er for eksempel ikke tillatt å kalle en metametode i ytterste nivå av en klassedefinisjon. Dessuten er tillatt syntaks for de aktuelle parametrene i et metametodekall akkurat den samme som for parametre i et vanlig metodekall. Det betyr i praksis at parametrene til en metametode må være uttrykk, det vil si at de svarer til subtrær i AST-et med rotnode av typen `Expression`. Det kunne for eksempel være fristende å prøve kalle en metametode med en klassedefinisjon som parameter, men det er altså ikke lovlig.

Hvis vi ser nærmere på hvordan metametoden i figur 3.6 er implementert, så er selve metodedefinisjonen ganske kort og den benytter seg av et generelt visitor-programmeringsmønster, slik det beskrives i [15], blant annet ved at den støtter seg til hjelpeklassen `SimplifyArithIdentityVisitor`. Det første metametoden gjør er å opprette en node av typen `ExpressionStatement` for uttrykket `expr` gitt som parameter til metoden. Dette er et eksempel på

hvordan nye AST-noder kan opprettes i en metametode. Årsaken til at vi legger denne noden over det opprinnelige uttrykket er en detalj som har å gjøre med hvordan `SimplifyArithIdentityVisitor` virker.

Nodene i AST-et støtter visitor-mønsteret ved at alle noder har en metode `Accept` som kan kalles med et visitor-objekt som parameter. Klassen `DepthFirstTransformer` er en av mekanismene i Boo for å manipulere AST-et, og den kan brukes til å opprette et visitor-objekt for å gjøre en dybde-først traversering av hele eller deler av AST-et.

Klassen `SimplifyArithIdentityVisitor` bygger på denne funksjonaliteten ved at den er en subklasse av `DepthFirstTransformer`, og brukes til å opprette et visitor-objekt for uttrykket som skal forenkles. Denne klassen inneholder derfor spesiell kode for de forenklingene som gjøres. Siden forenklingene her kun gjelder binære uttrykk (addisjon og multiplikasjon) definerer klassen kun en metode for slike uttrykk, `OnBinaryExpression`. Dersom et binæruttrykk tilfredsstiller betingelsene for at en forenkling kan gjennomføres, vil det endres (det vil si erstattes med den operanden som ikke er identitetselement). Denne metoden er en av de påkrevde metodene i grensesnittet for et visitor-objekt. Påkrevde metoder for andre nodetyper i AST-et arves fra `DepthFirstTransformer` siden de ikke trenger spesialtilpasning.

Annotasjon av noder i AST-et

Til slutt bør det også nevnes at nodene i AST-et kan gis annotasjoner. En annotasjon er en vilkårlig verdi som kan knyttes til en node med en nøkkelverdi. En node kan ha flere slike annotasjoner. Disse annotasjonene lagres i en hash-tabell i noden, og tilordning og oppslag av annotasjonene kan gjøres ved å referere direkte til noden:

```
node.Right["optimized"] = true
```

Denne tilordningen er hentet fra eksemplet i figur 3.6 og viser hvordan en node som har blitt optimalisert kan annoteres med verdien `true` for å indikere at den har blitt optimalisert.

Nytteverdien til akkurat dette eksemplet på annotasjoner er kanskje noe begrenset, men vi kan for eksempel tenke oss at de kan brukes i forbindelse med senere steg i kompilatorsekvensen for å tilrettelegge for testutskrifter eller feilsøking.

Annotasjoner er imidlertid svært nyttige til å organisere en samhandling mellom de forskjellige metaprogrammeringsmekanismene, for eksempel mellom to makroer eller et syntaktisk attributt og en makro. En makro kan for eksempel annotere en node i AST-et og denne annotasjonen kan slås opp av en annen makro.

3.3.2 Makroer

En makro i Boo ligner litt på en metametode, men makroer er på mange måter mer fleksible. I likhet med metametoder utføres makroer i løpet av kompileringen. En makro kan ha parametre på samme måte som en metametode, og en makro returnerer et subtre som settes inn i AST-et ved at det erstatter makrokallet. Dette kalles å ekspandere makroen.

Figur 3.7 viser et eksempel på en makrodefinisjon. Denne makroen er et eksempel på hvordan man relativt enkelt kan definere en ny løkkekonstruksjon i Boo. Makroen i figur 3.7 gir oss en variant av en `for`-løkke som er lik den man for eksempel finner i C#. (Boo sin vanlige `for`-setning ligner mer på det som kalles `foreach` i C# og en del andre språk.)

```
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast

class ForMacro(AbstractAstMacro):

  def Expand(macro as MacroStatement):
    if macro.Arguments.Count != 3:
      raise "For: three parameters are required"

    initStatement as Expression = macro.Arguments[0]
    condStatement as Expression = macro.Arguments[1]
    incrStatement as Expression = macro.Arguments[2]

    body = macro.Body
    return [|
      $initStatement
      while $condStatement:
        $body
        $incrStatement
    |]
```

Figur 3.7: Makrodefinisjon av en alternativ `for`-løkke

Makroen kan kalles på følgende måte:

```
For i = 1, i < 5, i++:
  j = 2 * i
  print i, j
```

Det er ganske slående hvor mye dette ligner på en vanlig konstruksjon i språket. Det eneste som egentlig gjør at det skiller seg litt ut er at makronavnet angis med stor forbokstav. Men dette er kun nødvendig i dette spesielle tilfellet ettersom `for` er et reservert ord for den vanlige `for`-løkkekonstruksjonen i Boo. Normalt skiller faktisk Boo ikke mellom store og små bokstaver i makronavn, så vi kunne til og med bruke `FOR` til å kalle denne makroen. Dette er i seg selv et pussig unntak i forhold til hvordan alle andre

identifikatortyper behandles i Boo, slik som navn på variabler, metoder eller klasser. For i disse er forskjell mellom store og små bokstaver signifikant.

Det bør også bemerkes at makroen i figur 3.7 riktignok gir oss en noe forenklet utgave av en `for`-løkke, og vil for eksempel ikke håndtere kall til `continue` inne i løkka på riktig måte, men med en litt mer avansert implementasjon av makroen er det også mulig.

En makro kan ha parametere og en kropp. Kroppen kan altså betraktes som en litt spesiell parameter. `For`-makroen over kalles for eksempel med tre parametre: `i = 1`, `i < 5` og `i++`, og en kropp som består av to setninger: en tilordning og en utskriftssetning. Parametrene til en makro må være uttrykk og ligner mye på metametode-parametre. En liten forskjell er at de aktuelle parametrene i et metametodekall må omsluttes med parenteser, mens det omvendte er tilfellet for makrokall. Her skal det ikke brukes parenteser. Kroppen til makroen angis på samme måte som en vanlig blokk i språket, det vil si med kolon, linjeskift og innrykk. Innholdet i kroppen er ikke begrenset til enkle uttrykk slik som parametrene er, men kan i tillegg bestå av en blokk med vanlige setninger, som i eksemplet over, og til og med metodedefinisjoner. Det er imidlertid ikke tillatt med klassedefinisjoner i kroppen til en makro. Det er som vi skal se en noe ugunstig begrensning som er relevant i forbindelse med vår implementasjon av BooPT.

Både parametrene og kroppen er i utgangspunktet valgfrie, men makrodefinisjonen kan legge føringer på antall parametre og hvorvidt det skal angis en kropp eller ikke. Dersom makroen ikke har en kropp skal det heller ikke settes kolon etter makrokallet. I vårt eksempel krever makrodefinisjonen at det angis nøyaktig tre parametre, men kroppen er faktisk valgfri. Vi kan altså lage en løkke uten kropp (noe som ikke er tillat i de vanlige Boo-løkkene):

```
For i = 1, i < 5, System.Console.WriteLine(i++)
```

Vi skal se nærmere på hvordan dette har seg, når vi nå ser mer på selve definisjonen av makroen. En makro defineres altså med en subklasse av typen `AbstractAstMacro`. Selve funksjonaliteten i makroen må defineres ved å implementere metoden `Expand`, som må returnere det som skal settes inn i AST-et isteden for makrokallet. Denne metoden har en parameter `macro` av typen `MacroStatement` som blant annet inneholder referanser til de aktuelle parametrene til makroen i `macro.Arguments` og kroppen i `macro.Body`.

Quasi-quotation

Tanken bak `For`-makroen i eksemplet over er at makrokallet skal gjøres om til en `while`-løkke, som er en faktisk løkkekonstruksjon i Boo. For å få til dette benytter makroen en mekanisme kalt *quasi-quotation*. Med quasi-quotation omsluttes Boo-kode med de spesielle parenteskonstruksjonene `[|` og `|]`.

Kode som forekommer mellom slike blir tolket på en spesiell måte av kompilatoren, og blir ikke umiddelbart utført når den forekommer for eksempel i en makro. Kompilatoren vil bygge AST-et for slik kode, men isteden for å oversette den til kjørbar kode, blir AST-et bevart og satt inn i programmet der quasi-quotation-koden forekommer. Dette er en nyttig teknikk som kan brukes av metametoder, makroer eller syntaktiske attributter til å generere nye deler av AST-et som disse operer på. Quasi-quotation-kode vil altså i det fleste tilfeller til slutt utgjøre en del av det ferdigkompilerte programmet.

I figur 3.7 ser vi hvordan quasi-quotation er brukt til å lage den nevnte `while`-løkka. Vi ser også hvordan vi kan angi substitusjoner i quasi-quotation-kode ved å bruke dollartegn etterfulgt av et variabelnavn. Slike variabler må referere til AST-subtrær og vil settes inn i AST-et som blir konstruert fra quasi-quotation-koden.

Alternativet til quasi-quotation er å konstruere AST-et ved å eksplisitt lage objekter for nodene som skal være en del av AST-et:

```
innerBlock = Block()
innerBlock.Add(body)
innerBlock.Add(incrStatement)
outerBlock = Block()
outerBlock.Add(initStatement)
outerBlock.Add(WhileStatement(condStatement, innerBlock)
```

Dette er både mer tungvint og mindre lesbart, så quasi-quotation er en kraftig mekanisme som gjør det enkelt for metametoder, makroer eller syntaktiske attributter å generere ny kode. Vi vil imidlertid i liten grad benytte oss av quasi-quotation videre i denne oppgaven, siden de makroer og syntaktiske attributter som utgjør implementasjonen av BooPT i liten grad brukes til å generere ny kode.

Mer om makroer

En potensielt nyttig egenskap ved makroer er at de tillates å ha sideeffekter på AST-et. De kan altså brukes til å modifisere andre deler av AST-et enn det som er tilgjengelig gjennom parametrene og kroppen til makroen. Man kan for eksempel få tilgang til andre noder i AST-et ved å følge pekeren `macro.ParentNode`. Makroen i vårt eksempel har imidlertid ingen slike sideeffekter.

Det finnes også en annen måte å definere makroer på. Boo har nemlig innebygd en egen makro kalt `macro` for definisjon av andre makroer. Bruk av denne gjør en makrodefinsjon noe mer kompakt, siden `macro`-makroen automatisk oppretter en klasse for makroen slik at det kun er innholdet i `Expand`-metoden som må angis av programmereren. Eksempler på dette er gitt i boken til Rahien[8].

Makrokall kan dessuten nøstes, ved at makrokall kan forekomme i kroppen til en annen makro. Dette er en nyttig teknikk som også er nærmere beskrevet i [8]. Det har imidlertid ikke vært nødvendig å benytte seg av dette i forbindelse med implementasjonen av BooPT.

3.3.3 Syntaktiske attributter

Et attributt kan beskrives som en mekanisme for å knytte informasjon til konstruksjoner i språket[16]. Vi skiller mellom to typer attributter i Boo, *vanlige attributter* og *syntaktiske attributter*, selv om det er vanlig å bare bruke betegnelsen "attributter" om de første. Vanlige attributter er svært like C#-attributter, og Boo-programmer kan for eksempel benytte seg av attributter definert i .NET sitt Base Class Library. Boo sine attributter tilsvarer det som kalles annotasjoner i Java.

Hovedfokus for denne seksjonen skal være på syntaktiske attributter og mulighetene for metaprogrammering de gir oss, men siden et syntaktisk attributt kan sees som et spesialtilfelle av vanlige attributter, er det til å begynne med også nyttig å se litt på likheter og forskjeller mellom disse to typene.

Felles for begge typer attributter er syntaksen for hvordan de brukes:

```
[singleton]
class Test:
  pass
```

(Nøkkelordet `pass` brukes for å angi tomme blokker i Boo, og er brukt ettersom det i dette eksemplet ikke er viktig hva klassen faktisk gjør.) Dette eksemplet angir attributtet `singleton` for klassen `Test`. Attributtnavnet omsluttet av hakeparenteser må altså settes før programkonstruksjonen det skal gjelde for, i dette tilfellet en klasse. Dette bestemte attributtet angir at singleton-programmeringsmønsteret[15] skal gjelde for denne klassen. En klasse med dette attributtet kan altså kun instansieres en gang. Slik som med makroer skilles det ikke mellom store og små bokstaver i et attributtnavn. Vi kunne altså like gjerne skrevet for eksempel `[Singelton]`. Vi har dessuten allerede sett et annet eksempel på bruk av attributter, i forbindelse med definisjon av metametoder. Der brukte vi attributtet `meta` til å angi at metoden med dette attributtet skal tolkes spesielt av kompilatoren, altså som en metametode.

Attributter kan generelt brukes på forskjellige typer konstruksjoner i et program. Dette inkluderer klasser og andre typedefinisjoner, klassemedlemmer, vanlige metodedefinisjoner og parametre. Attributter kan imidlertid ikke brukes på enkle programsetninger eller lokale variabeldeklarasjoner, for eksempel. Implementasjonen av et spesielt attributt kan dessuten legge begrensninger på hvilke typer konstruksjoner det kan brukes på. Attributtet

singleton kan for eksempel kun brukes på klasser mens meta kun kan brukes på metodedefinisjoner.

Attributter kan også ha parametre. Når et attributt anvendes må de aktuelle parametrene angis i parentes bak attributtnavnet, men innenfor hakeparentesene. Som vanlig brukes komma for å skille parametrene fra hverandre. Ingen av de to attributtene singleton og meta har noen parametre.

Attributtene singleton og meta er for øvrig eksempler på de to forskjellige attributtypene, henholdsvis et syntaktisk og et vanlig attributt. Denne forskjellen er imidlertid ikke opplagt ved vanlig bruk av attributtene, om man ikke også studerer hvordan attributtene er definert. Begge disse attributtene er imidlertid innebygget i språket.

Både vanlige og syntaktiske attributter defineres ved hjelp av klasser. Et vanlig attributt defineres med en subklasse av klassen System.Attribute fra BCL. Attributter, både vanlige og syntaktiske, brukes typisk til å påvirke kompileringen av programmet på en eller annen måte. Men et vanlig, egendefinert attributt vil ikke være kjent for kompilatoren, så et slikt vil ikke kunne brukes til å påvirke kompilatoren på noen som helst måte. Vanlige attributter kan imidlertid brukes til andre formål. Et program kan for eksempel undersøke hvilke attributter forskjellige elementer i programmet har mens det kjører, ved hjelp av refleksjon[17].

Et syntaktisk attributt defineres med en subklasse av klassen Abstract-AstAttriubte fra navnerommet Boo.Lang.Compiler. Et syntaktisk attributt skiller seg fra et vanlig attributt ved at det er mulig å definere en metode som skal utføres når dette attributtet oppdages under kompileringen, og denne metoden kan brukes til å manipulere AST-et på forskjellig vis. Figur 3.8 viser et nokså trivielt eksempel på definisjon av et syntaktisk attributt.

Tanken med dette attributtet er at det kan brukes til å fjerne programkonstruksjoner fra den videre kompileringen, slik at det ikke genereres noe kode for disse konstruksjonene. Samtidig gir det mulighet for å skrive ut en forklarende melding, for eksempel om hvorfor den aktuelle konstruksjonen er unnlatt fra kompilering. Dette kan altså sees på som en litt avansert form for å kommentere ut kode. Koden med dette attributtet blir imidlertid kjørt gjennom parseren, så den må være syntaktisk korrekt, ellers vil kompileringen feile. Følgende eksempel viser hvordan attributtet kan brukes på klassen Test over:

```
[skip("not implemented: class Test")] [singleton]
class Test:
  pass
```

Dette viser også at det er lov å bruke flere attributter på samme programkonstruksjon. Hvis vi ser nærmere på hvordan attributtet er definert i klassen SkipAttribute, ser vi at den har tre konstruktører. Parametrene til disse

```
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast


class SkipAttribute(AbstractAstAttribute):
  _message as string = null
  _doprint as bool = true


  def constructor():
    pass

  def constructor(messageParam as StringLiteralExpression):
    _message = messageParam.Value

  def constructor(doprintParam as BoolLiteralExpression):
    _doprint = doprintParam.Value

  def Apply(node as Node):
    if node isa ParameterDeclaration:
      raise "cannot use skip on parameter declarations"

    member as TypeMember = node
    parent as TypeDefinition = node.ParentNode
    parent.Members.Remove(member)

    if not _doprint:
      return
    if _message is null:
      print "skipping compilation of:", member.Name
    else:
      print _message
```

Figur 3.8: Definisjonen av attributtet `skip`

konstruktørene svarer til de lovlige parametrene for attributtet. Siden `skip`-attributtet har tre konstruktører kan det altså brukes på tre forskjellige måter, enten uten parametre, med en tekststreng som parameter eller med en boolsk verdi som parameter. For å angi at attributtet skal kunne brukes uten parameter måtte vi i dette tilfellet eksplisitt definere konstruktør uten parameter, selv om denne ikke gjør noen ting. Dette er fordi det er definert konstruktører med parametre. Dersom det ikke er definert noen konstruktører for et attributt vil det angi at attributtet kun kan brukes uten parameter.

Metoden `Apply` inneholder koden som blir utført når attributtet behandles i løpet av kompileringen. Det er altså denne som skiller syntaktiske fra vanlige attributter og gjør dem til en spesielt kraftig mekanisme. Noden i AST-et som svarer til konstruksjonen som attributtet er anvendt på blir gitt som

aktuell parameter til denne metoden. For eksemplet med `Test` er dette altså en klassedefinisjonsnode. Vi ser eksempel på hvordan metoden modifiserer syntakstreet ved at den sletter denne noden fra listen over medlemmer i foreldrenoden. (Foreldrenoden er i vårt eksempel en modul som kan ha klasser og andre typedefinisjoner som medlemmer.) Syntaktiske attributter kan ikke ha noen returverdi, i motsetning til metametoder og makroer.

Videre ser vi at `skip`-attributtet kan brukes på alle mulige konstruksjoner som tillates å ha et attributt, med unntak av parameterdeklarasjoner. Dersom metoden `Apply` oppdager at attributtet er brukt på en parameterdeklarasjon, så kaster den et unntak.

Både syntaktiske attributter og makroer blir behandlet i kompilatorsteget `MacroAndAttributeExpansion`, men alle syntaktiske attributter blir behandlet før makroene. For øvrig behandles attributter i en rekkefølge gitt ved en dybde-først traversering av syntakstreet. Det samme gjelder for makroer. Dette er noe man eventuelt må ta hensyn til i forbindelse med implementasjon av slike makroer og attributter, spesielt dersom skal være noen form for samhandling mellom dem. Det er da viktig å være klar over denne detaljen, som jo legger visse føringer for hvordan slik samhandling kan foregå. Dette er for eksempel noe vi utnytter i vår implementasjon av BooPT, som vi skal se mer på senere.

Kapittel 4

Boo Package Templates

Et viktig mål med denne oppgaven har vært å vurdere hvor sterke og fleksible Boo sine metaprogrammeringsmekanismer er, og å undersøke hvor godt de egner seg til å implementere PT-mekanismen for Boo.

Det har også vært ønskelig å forsøke å få til dette uten å måtte gjøre endringer direkte i Boo-kompilatoren og heller ikke gripe for mye inn i kompilatoren og kompilatorsekvensen utover det muligheter for dette som tilbys gjennom metametoder, makroer og syntaktiske attributter.

Vi skal se at dette langt på vei er mulig, og jeg vil derfor beskrive en slik implementasjon av de mest sentrale elementene i PT for Boo, som jeg altså har valgt å kalle BooPT.

Et annet delmål har vært å forsøke å oppnå en syntaks for BooPT som ligger tett opp til den opprinnelige PT-syntaksen beskrevet i kapittel 2, men som likevel er tilpasset språket Boo.

BooPT er altså hovedfokus for kapitlet. Pakker er et viktig begrep i PT-mekanismen, så vi starter med en diskusjon av hvordan dette er relatert til Boo og BooPT. Videre gis en kort introduksjon til BooPT og hvilken syntaks som brukes i BooPT. Det gis også et enkelt eksempel. Deretter gis en relativt detaljert forklaring av hvordan BooPT er implementert. Dette inkluderer en vurdering av hvilke metaprogrammeringsmekanismer i Boo som egner seg for å implementere de ulike konstruksjonene i PT, en begrunnelse av syntaksen som ble valgt for BooPT og hvilke valg som ellers ble gjort i forbindelse implementasjonen.

Til slutt vurderer jeg i hvilken grad Boo har vært egnet til å lage en enkel variant av PT, slik som BooPT er. Dette inkluderer blant annet beskrivelse av enkelte begrensinger i Boo som har lagt føringer på implementasjonen av BooPT, eller gjort den mer komplisert enn det som er ønskelig.

4.1 Pakker i Boo

En pakke er et sentralt konsept som PT-mekanismen bygger på, så det er nyttig å avklare hvordan dette kan relateres til Boo. Som vi var inne på i innledningen, så støtter de fleste objektorienterte språk et slikt konsept ved at de har en mekanisme for å samle klasser i avgrensede enheter. Når vi utformer støtte for PT for et bestemt språk vil det derfor være naturlig å ta utgangspunkt i den eksisterende mekanismen for pakker i dette språket.

Boo støtter også et slikt konsept, men bruker en annen betegnelse enn pakke. I Boo grupperes klasser i såkalte navnerom, som nok er det som passer best med begrepet pakke slik det brukes i PT.

Men i Boo deles også klasser inn på andre måter enn bare ved hjelp av navnerom. På kildekodenivå grupperes klasser i moduler, som bare er et annet navn for filer som inneholder Boo-kildekode. En slik modul kan inneholde en eller flere klasser. Etter kompilering til bytekode grupperes klassene i .NET-assemblier. Flere moduler kan kompileres til ett .NET-assembly.

Et navnerom kan spenne over flere Boo-moduler, og til og med flere .NET-assembly. Moduler og .NET-assembly er altså mekanismer som er beregnet for inndeling av klasser på henholdsvis kilde- og bytekodenivå, mens et navnerom kan sees som en mer konseptuell inndeling.

Når vi innfører PT i Boo ville det altså være naturlig at en template blir å regne som et navnerom med de forbedringer og utvidelser som PT-mekanismen medfører. På den måten ville klassene i en template kunne spres over flere moduler. For å gjøre implementasjonen noe enklere har jeg imidlertid begrenset meg til at en template kun kan omfatte klassene i én enkelt modul. Slik sett blir en template i BooPT mer å regne som en spesiell modul som har de egenskapene som er beskrevet i seksjon 2.

Det ville antagelig ikke være så vanskelig å utvide eller forbedre implementasjonen slik at en template i større grad samsvarer med navnerombegrepet.

En template i BooPT er altså tett relatert til en Boo-modul, så det blir et viktig begrep når vi skal gå gjennom implementasjonen av BooPT.

4.2 Bruk av BooPT

Denne seksjonen gir en kort introduksjon til hvordan BooPT vil se ut for en programmerer som benytter seg av denne mekanismen. Det fokuseres altså først og fremst på syntaksen for å definere og bruke templater. En forklaring av betydningen av de forskjellige konstruksjonene i PT er allerede gitt i seksjon 2.

Funksjonaliteten i PT-implementasjonen er samlet i navnerommet kalt `BooPT`,

så alle programmer som definerer eller bruker templater må importere dette navnerommet på følgende måte: `import BooPT`.

Ved kompileringen av slike programmer må dessuten .NET-assembliet med dette navnerommet kobles inn. For å kompilere en template i filen `Template.boo` brukes derfor typisk følgende kommando: `booc -r:BooPT.dll Template.boo`. For å kompilere et program som instansierer en template gjøres følgende: `booc -r:BooPT.dll -r:Template.dll program.boo`. Dersom en template selv instansierer andre templater må også assembliene for disse angis med `-r` (i den første kommandoen).

4.2.1 Deklarasjon av templater

En template deklareres ved å angi ordet `template` etterfulgt av navnet templaten skal ha. Dette er egentlig et makrokall, men det skal vi komme tilbake til.

Denne deklarasjonen må gjøres på ytterste nivå i en Boo-modul, og angir at alle klassene i modulen skal være en del av templaten. En modul kan inneholde kun én template-deklarasjon. Deklarasjonen kan i utgangspunktet forekomme hvor som helst i modulen etter `import`-setningene, men det er naturlig at den settes inn et sted i begynnelsen av modulen. Figur 4.1 er et konkret eksempel på hvordan graf-templaten fra seksjon 2 kan defineres i BooPT.

Når denne templaten kompileres blir resultatet et vanlig .NET-assembly som kan brukes av Boo-programmer som ønsker å instansiere templaten.

Syntaksen for template-deklarasjon skiller seg noe fra den opprinnelige PT-syntaksen beskrevet i seksjon 2. For det første, benyttes ingen klammeparenteser for å avgrense innholdet i template-definisjonen. Dette er naturlig siden Boo fortrinnsvis bruker innrykk for å avgrense blokker. På grunn av begrensinger i Boo er det imidlertid ikke mulig å bruke slike innrykk for å avgrense template-deklarasjoner. Derfor vil som nevnt en template-deklarasjon gjelde hele modulen der den forekommer.

4.2.2 Instansiering av templater

En template instansieres ved å bruke ordet `inst` etterfulgt av navnet på templaten som skal instansieres og eventuelle omnavninger av klasser og klassemedlemmer i templaten. Slik som `template` er `inst` egentlig et makrokall, noe vi kommer tilbake til i seksjon 4.3. En slik instansiering må angis på ytterste nivå i modulen og kan forekomme hvor som helst i modulen.

```
import BooPT

inst Graph(Node >> Junction(firstEdge >> firstRoad, \
```

```
import BooPT

template Graph

class Node:
  firstEdge as Edge = null
  lastEdge as Edge = null

  def insertEdge(to as Node) as Edge:
    edge = Edge(self, to)
    if firstEdge is null:
      firstEdge = edge
    else:
      lastEdge.nextEdge = edge
      edge.prevEdge = lastEdge
    lastEdge = edge
    return edge

class Edge:
  start as Node
  end as Node
  internal prevEdge as Edge = null
  internal nextEdge as Edge = null

  def constructor(start as Node, end as Node):
    self.start = start
    self.end = end

  def removeMe():
    prevEdge.nextEdge = nextEdge
```

Figur 4.1: Template for en graf-struktur

```
                        lastEdge >> lastRoad, \
                        insertEdge(Node) >> insertRoad), \
           Edge >> Road(prevEdge >> prevRoad, \
                        nextEdge >> nextRoad))

j1 = Junction()
j2 = Junction()
road = j1.insertRoad(j2)
```

Vi ser at syntaksen til `inst` ligner mye på den opprinnelige PT-syntaksen fra kapittel 2, men det er et par forskjeller. For det første brukes ikke ordet `with` for å introdusere omnavninger som angis etter template-navnet. Omnavningene skilles isteden fra template-navnet ved å bruke parenteser. Dessuten brukes ikke `=>` og `->` for å angi omnavning av henholdsvis klasser og medlemmer av klassene. Isteden brukes `>>` for begge typer omnavning. Gjeldende utgave av Boo tillater nemlig ikke definisjon av nye operatorer, så derfor må vi bruke en eksisterende operator til dette formålet, og operato-

ren for bitskifting mot høyre er kanskje den som utseendemessig passer best. Men betydningen av operatoren er altså noe helt annet her enn ellers i Boo. Vi ser dessuten her eksempel på at slike `inst`-setninger kan bli lange. Siden setninger i Boo avsluttes med et vanlig linjeskift, så må vi, om vi ønsker å dele en setning inn i flere linjer, bruke et spesielt symbol (\) for å angi at setningen fortsetter på neste linje.

Utvidelser av klasser fra en instansiert template kan angis med attributtet `adds` i forbindelse med en klassedeklarasjon. Klassedeklarasjonen må ha samme navn som den instansierte template-klassen. Attributtet angir altså at dette ikke er en vanlig klassedeklarasjon, og den vil således ikke gi opphav til en egen klasse. Medlemmene i klassedeklarasjonen vil derimot legges til den template-klassen. Her er et eksempel på hvordan klassene `Junction` og `Road` kan utvides med nye felt og metoder:

```
[adds]
class Junction:
  internal junctionNo as int
  internal trafficLights as bool

  def insertRoad(to as Junction):
    road = tsuper.insertRoad(to)
    road.oneway = false
    road.roadType = "highway"
    return road

  def insertOnewayRoad(to as Junction, roadType as string):
    road = tsuper.insertRoad(to)
    road.oneway = true
    road.roadType = roadType

  def insertDualCarriagewayRoad(to as Junction):
    insertOnewayRoad(to, "highway")
    to.insertOnewayRoad(self, "highway")

[adds]
class Road:
  internal roadType as string
  internal oneway as bool
  internal lanes as int
```

Vi ser her at klassene `Junction` og `Road` utvides med henholdsvis feltene `junctionNo`, `trafficLights` og `roadType` og `oneway`. Klassen `Junction` får i tillegg en ny metode `insertOnewayRoad`, men vi ser også hvordan en eksisterende metode, `insertRoad`, kan redefineres i `adds`-klassen. Dessuten har ordet `tsuper` en spesiell betydning i `adds`-klassen, slik som forklart i seksjon 2.2. Dette gjør det mulig å få tilgang til den opprinnelige utgaven av en metode som har blitt redefinert i `adds`-klassen.

4.3 Implementasjon

Denne seksjonen gir en relativt detaljert beskrivelse av hvordan BooPT er implementert. De forskjellige konstruksjonene i PT er implementert kun ved bruk av makroer og et attributt. Vi har sett eksempler på syntaksen til de ulike konstruksjonene i BooPT i seksjon 4.2. Det er lett å se at `adds` er ett attributt, men kanskje ikke like opplagt at bruk av `template` og `inst` faktisk er makrokall.

Boo har for øyeblikket ikke støtte for å utvide syntaksen i språket utover det er som er mulig syntaktiske attributter og makroer. Det er derfor først og fremst betraktninger i forhold til hvilken syntaks som er mest passende som har vært avgjørende for om det har blitt benyttet en makro eller et syntaktisk attributt for å implementere de ulike konstruksjonene.

Bortsett fra de syntaktiske forskjellene er makroer og syntaktiske attributter stort sett like kraftige mekanismer for metaprogrammering. Det har ikke vært aktuelt å bruke metametoder i implementasjonen av PT. Metametoder er stort sett kun nyttige dersom man ønsker å behandle mindre biter av et program, slik som et uttrykk eller deler av et uttrykk. Utover at de har denne muligheten er de heller ikke like fleksible som makroer eller attributter. Konstruksjonene i PT brukes på et syntaktisk mer overordnet nivå, det vil si i forbindelse med deklarasjon av pakker og klasser, slik at den fordelen metametoder har i forhold til makroer eller syntaktiske attributter ikke er relevant i denne sammenhengen.

Vi skal under se mer detaljert på hvordan `template`, `inst` og `adds` er implementert. Av disse er det i stor grad `inst` som gjør det meste av jobben. Oppgaven til `adds`-attributtet er hovedsakelig å registrere en `adds`-klasse og legge til rette for at den senere kan behandles av `inst`.

Tilsvarende blir oppgaven til `template`-makroen å registrere innholdet i en template og lagre denne på en form som senere kan brukes av `inst`. En template deklareres og kompileres separat fra programmer der den kan instansieres. En sentral utfordring i denne oppgaven har vært å finne et passende format for å gjøre en kompilert template tilgjengelig i programmet der den skal instansieres. Vi har valgt å bruke et vanlig .NET-assembly til dette formålet, men innholdet i dette vil skille seg litt fra skiller fra et typisk .NET-assembly. Den vil nemlig ikke inneholde bytekode, men isteden en tekstlig representasjon av koden til templaten. Når denne templaten instansieres, vil `inst` ha tilgang til den tekstlige koden for templaten dersom riktig .NET-assembly kobles inn når programmet som gjør instansieringen kompileres. Dette kan umiddelbart virke som en litt pussig løsning. Grunnen til dette valget og mulige alternativer blir diskutert i seksjon 4.4.2.

Koden for implementasjonen er relativt kort, så den er gjengitt i tillegg A.

Forklaringen som følger krever ikke at man også studerer denne koden, men det kan være ønskelig for å få en enda bedre forståelse av implementasjonen.

4.3.1 `template`

Vi har valgt å implementere template-deklarasjon med en makro. Vi starter med en begrunnelse av hvorfor vi har valgt en makro for dette formålet, og ser deretter mer detaljert på hvordan makroen faktisk er implementert.

Vi forklarte ovenfor hvorfor en metametode ikke egner seg for konstruksjonene i BooPT. En template-deklarasjon gjelder en samling av klasser, så et syntaktisk attributt egner seg heller ikke, siden en forekomst av et attributt kun kan angis for enkeltelementer i programmet. Ønsker man for eksempel at et attributt skal gjelde for mer enn bare en klasse, må attributtet gjentas for alle klassene. Skulle vi derfor bruke et attributt til å deklarere en template `MyTemplate`, måtte vi gjentatt noe slikt som `[template(MyTemplate)]` for alle klasser vi ønsker å ha med i templaten. Dette kunne kanskje vært en mulighet, men den harmonerer dårlig med den tradisjonelle PT-syntaksen beskrevet i kapittel 2.

Syntaktisk sett er det antagelig derfor en makro som er den mest passende mekanismen i Boo for implementere template-deklarasjon. For å rydde eventuell tvil av veien bør det bemerkes at templater ikke deklareres ved å definere en egen makro for hver template. Vi definerer derimot en generell makro, kalt `template`, og vi bruker et kall til denne hver gang vi skal deklarere en ny template.

Nå ser vi nærmere på hvordan `template`-makroen er implementert. Hovedpoenget med `template`-makroen er, som nevnt innledningsvis, å sørge for at når en template blir kompilert, så skal den ikke kompileres til bytekode på vanlig måte, men at koden for templaten isteden skal bli lagret som tekst i et .NET-assembly som er resultatet av kompileringen. Når et program senere ønsker å instansiere templaten ved å bruke `inst`, hentes så template-koden fra dette .NET-assembliet og kompileres isteden som en del av dette programmet.

Denne `template`-makroen har kun én parameter, template-navnet. Et kall til denne makroen skal som nevnt i seksjon 4.2.1 angi at alle klasser i modulen der kallet forekommer skal være en del av templaten. Dette løses ved at makroen gjennom sin `ParentNode`-peker har tilgang til noden i AST-et til denne modulen, og dermed også til nodene for alle klassene som skal være en del av templaten.

Makroen benytter seg av mulighetene som Boo gir for å modifisere AST-et. Det første som blant annet gjøres er at selve kallet til `template`-makroen slettes fra modulen. Denne modulen inneholder alle klassene i template-

deklarasjonen og disse skal settes inn i programmet der templaten instansieres, men selve `template`-kallet, som angir at denne modulen skal tolkes som en template, må ikke settes inn. Derfor slettes det fra modulen ved å modifisere AST-et.

Intern kompilering av templaten

Makroer utføres, som nevnt i seksjon 3.2.1, i kompilatorsteget `MacroAndAttributeExpansion`. Dette steget forekommer relativt tidlig i kompilatorsekvensen, så `template`-makroen blir utført når relativt mye av kompileringen gjenstår. Idéen med denne makroen er å opprette en klasse med et felt `_templateCode` som inneholder en tekstlig representasjon av template-koden. Det er imidlertid ønskelig at template-koden kjøres gjennom hele kompilatoren før den lagres i dette feltet, slik at blant annet all feilsjekking av template-koden utføres. Etter at makroen har opprettet klassen med feltet med template-koden, slettes nemlig alle andre klasser i denne modulen fra AST-et. Disse klassene, som altså er klasser i templaten, blir bevart som tekstlig kode når de lagres i det nevnte feltet. Det er altså ingen grunn til å la disse være en del av det .NET-assembly som er resultatet av kompileringen. Om man tillot dette kunne det gi uønskede konsekvenser om et program som bruker et slikt assembly har en uheldig oppførsel og forsøker å få tilgang til template-klassene direkte, uten å bruke `inst` slik det er meningen.

Template-klassene vil isteden bli gjort tilgjengelig etter at en template er instansiert med `inst`. Disse vil derfor slettes og ikke behandles videre i den vanlige kompilatorsekvensen etter at steget `MacroAndAttributeExpansion` er utført.

Denne makroen oppretter derfor en egen, intern instans av Boo-kompilatoren for å kompilere template-koden, inkludert disse klassene. Dette er altså en instans av Boo-kompilatoren som kjøres (rekursivt) i løpet den vanlige kompileringen. Dette gjøres av to grunner. Den ene er som nevnt at templaten bør sjekkes for feil før den tas i bruk. Det andre er at kompileringen vil resultere i et noe forenklet AST som kun består av klasser. For eksempel vil eventuelle `import`-setninger i templaten gjøres overflødige ved at Boo-kompilatoren gjør alle referanser til importerte navnerom eksplisitte. Dessuten vil alle metametoder, makroer og attributter i templaten utføres. Dette er viktig siden en template selv kan instansiere andre templater, og det gjøres ved hjelp av `inst`, som er en makro. *Slik instansiering av andre templater vil altså utføres i forbindelse med denne interne kompileringen.*

Input til dette interne kompilatorkallet er altså den tekstlige template-koden, og den blir gitt som en tekststreng til kompilatoren ved å kalle metoden `ToCodeString` på noden i AST-et som svarer til modulen som inneholder template-klassene. Output er AST-et som er resultatet av siste steg i kom-

pileringen, og ved hjelp av `ToCodeString` får vi den tilsvarende koden som skal lagres i feltet felt `_templateCode`.

Kompilatorsekvensen `CompileWithTranslateGeneratedNamesStep`

Denne interne kompileringen gjør altså ingen kodegenerering, siden vi ikke er interessert i bytekode, men i AST-et og den tilsvarende koden som er resultatet av kompilatorstegene som kommer før kodegenereringen. Til denne kompileringen bruker vi derfor ikke standard-kompilatorsekvensen til Boo, `CompileToFile`, men en kompilatorsekvens basert på `Compile`. Kompilatorsekvensen `Compile` er en alternativ, innebygget kompilatorsekvens i Boo som gjør alt unntatt kodegenerering. Vi kan imidlertid dessverre ikke bruke denne kompilatorsekvensen helt uten videre. For, som nevnt i slutten av seksjon 3.2.2, kan det transformerte AST-et som er resultatet av en kompilering inneholde identifikatorer som inneholder dollartegn, og slike identifikatorer er ikke tillatt i vanlig Boo-kode. Dersom vi bruker metoden `ToCodeString` på et slikt AST-et får vi altså kode som parseren til Boo ikke vil akseptere. Så før vi kaller metoden `ToCodeString` må identifikatorer med dollartegn oversettes til lovlige identifikatorer, slik at resultatet blir gyldig Boo-kode. Dette er løst ved å utvide kompilatorsekvensen `Compile` med et ekstra steg, kalt `TranslateGeneratedNamesStep`, som utfører en slik oversettelse. Steget bruker en svært enkel algoritme som erstatter alle forekomster dollartegn i identifikatorer med et understrekingssymbol (_), som er et lovlig tegn i identifikatorer. Dette gjøres på en måte som sikrer at det oversatte identifikatornavnet ikke allerede er brukt i programmet. Det vil si, det legges til et suffiks om det er nødvendig. Vi har derfor definert en egen kompilatorsekvens `CompileWithTranslateGeneratedNamesStep` som svarer til `Compile` men som er utvidet med dette steget.

Resultatet av template-kompileringen

Vi bruker altså en vanlig tekstlig representasjon av template-koden innkapslet i et .NET-assembly for å gjøre templaten tilgjengelig for programmer der den kan instansieres. Template-koden lagres som verdien til et felt i en ny klasse som opprettes av makroen. Denne klassen blir altså en del av et .NET-assembly som er resultatet av kompileringen. For templaten `Graph` fra 4.1 vil følgende klasse opprettes:

```
class Graph:
  protected _templateCode as string = 'import BooPT\n\n
public class Node(object):\n\n\tprotected firstEdge as Edge\n\n
\tprotected lastEdge as Edge\n\n\tpublic def insertEdge(to as Node)
as Edge:\n\t\tedge = Edge(self, to)\n\t\tif self.firstEdge is null:\n
\t\t\tself.firstEdge = edge\n\t\telse:\n\t\t\tself.lastEdge.nextEdge
= edge\n\t\t\tedge.prevEdge = self.lastEdge\n\t\tself.lastEdge
```

```
= edge\n \t\treturn edge\n\n\tpublic def constructor():\n
\t\tsuper()\n\npublic class Edge(object):\n\n\tprotected start as
Node\n\n\tprotected end as Node\n\n\tinternal prevEdge as Edge\n\n
\tinternal nextEdge as Edge\n\n \tpublic def constructor(start as
Node, end as Node):\n\t\tsuper()\n\t\tself.start = start\n\t\tself.end
= end\n\n\tpublic def removeMe() as void:\n\t\tself.prevEdge.nextEdge
= self.nextEdge\n\n'
```

Vi ser at selve koden for templaten lagres i feltet `_templateCode`. Formateringen gjør ikke koden spesielt leselig for mennesker, men det er jo heller ikke meningen at det ved vanlig bruk av templater skal være nødvendig å studere denne koden. Når denne templaten instansieres med `inst`, vil `inst` hente koden ut fra dette feltet og kompilere den.

4.3.2 `adds`

Vi har implementert utvidelser av klasser fra en instansiert template ved hjelp av et syntaktisk attributt, `adds`. En utvidelse av en slik template-klasse er en samling av metoder og felt som klassen skal utvides med. Det er derfor naturlig å spesifisere en slik utvidelse med noe som ligner en vanlig klassedefinisjon og som kan inneholde både metoder og felt. For eksempel, med den originale PT-syntaksen så ser deklarasjonen av en utvidelse nesten ut som en vanlig klassedefinisjon, med det unntak at klassenavnet etterfølges av ordet `adds`. Men betydningen av en slik utvidelsesdeklarasjon er som vi vet forskjellig fra en vanlig klassedefinisjon. I BooPT deklareres utvidelser med en vanlig klassedefinisjon som merkes med det spesielle attributtet `adds`.

Det ville ikke være like naturlig å bruke en makro for å implementere slike utvidelser, siden, slik det ble nevnt over, Boo ikke tillater at kroppen til et makrokall inneholder klassedefinisjoner. Vi kunne imidlertid tenke oss en makro, kanskje kalt `class_adds` eller bare `adds`, som kunne kalles ved å liste opp alle utvidelser i form av funksjons- og variabeldeklarasjoner direkte i makrokroppen, altså uten en omsluttende klassedefinisjon. Hvorvidt dette ville gi en bedre syntaks eller ikke er antagelig en smakssak, men det ville også gi en mer teknisk komplisert implementasjon. Grunnen til det er at funksjons- og variabeldeklarasjoner i en makrokropp vil tolkes av Boo nettopp som generelle funksjons- og variabeldeklarasjoner. Disse representeres med andre typer noder i AST-et enn metode- og feltdeklarasjonene i en klasse, så for å innlemme slike i en template-klasse måtte AST-nodene først omformes til riktig nodetype. Ved å bruke et syntaktisk attributt til å implementere utvidelse unngår vi denne problemstillingen.

Dersom vi nå ser konkret på implementasjonen av `adds`-attributtet, er denne nokså kort og grei. Det som blir gjort er at klassedefinisjonen med dette attributtet blir registrert i en egen tabell for modulen som er under kom-

pilering. Denne tabellen, some jeg vil kalle for `adds`-tabellen, opprettes ved å bruke mekanismen for å annotere noder i AST-et. Denne mekanismen ble forklart nærmere i seksjon 3.2.2 Det er altså noden for modulen som er under kompilering som vil få en annotasjon kalt "adds" som vil peke på `adds`-tabellen. Etter dette er gjort vil klassedefinisjonen med utvidelsene slettes fra modulen, siden dette ikke er noen vanlig klasse som skal kunne brukes i programmet. (Klassedefinisjonen vil imidlertid fortsatt være tilgjengelig ved at `adds`-tabellen har en referanse til den.) All videre behandling av klassedefinisjonen med utvidelser vil gjøres i `inst`-makroen.

4.3.3 `inst`

Vi har også valgt å bruke en makro for å implementere `inst`. Dette er et naturlig valg siden `inst` kan betraktes som en ny setning i det utvidete språket som PT-syntaksen gir oss, og et makro-kall fremstår syntaktisk nettopp som en vanlig programsetning i Boo. Det er ikke naturlig å bruke et syntaktisk attributt til dette siden et slikt fremstår syntaktisk som en merkelapp som festes til andre konstruksjoner i språket, men `inst` skal behandles som en egen setning.

Implementasjonen av `inst`-makroen er relativt omfattende og består av selve klassen som definerer makroen `InstMacro`, og noen hjelpeklasser, `RenameReferenceVisitor`, `ResolveTSuperVisitor` og `RemoveTypeBindingsVisitor`. Klassen `InstMacro` har dessuten en rekke hjelpemetoder utover den påkrevde `Expand`-metoden. Utifra navnene på hjelpeklassene kjenner vi igjen at disse er visitor-klasser som kan brukes på AST-et. Disse brukes til å traversere og modifisere AST-et til templaten som skal instansieres på forskjellige måter, noe vi snart kommer tilbake til. Litt grovt kan virkemåten til `inst`-makroen oppsummeres som følger:

1 Bygging av omnavningstabeller

2 Initialisering

3 Finne templaten

4 Intern kompilering av templaten

5 Utføre omnavninger

6 Sletting av bindinger fra AST-et

7a Legge til utvidelser (adds-klasser)

7b Sammenslåing av instansierte klasser med samme navn

For hver av template-klassene vil kun ett av de to siste punktene blir utført. Dette avhenger av hvorvidt det allerede er instansiert en klasse med samme navn i forbindelse med instansieringen av en annen template.

Bygging av omnavningstabeller

Det første makroen gjør er å håndtere makro-parameteren. Denne må bestå av template-navnet, og eventuelt en mengde omnavninger av klasser og klassemedlemmer. Makroen aksepterer kun én parameter, selv om denne kan bestå av mange komponenter. Syntaktisk må parameteren fremstå som et uttrykk av sammensatte metodekall, identifikatorer og binære uttrykk, men `inst`-makroen tolker dette på en helt annen måte enn som et vanlig uttrykk. Parameteren som blir gitt til `inst`-makroen er altså delen av AST-et som svarer til dette uttrykket. Denne makroen traverserer dette, og fastslår navnet på templaten samt bygger omnavningstabeller på grunnlag av de metodenavn og identifikatorer som forekommer i uttrykket. Vi går nå mer detaljert gjennom hvordan dette blir gjort.

Vi bruker graf-templaten fra seksjon 4.2 som et eksempel. Denne kan for eksempel instansieres ved å bare gjøre en enkel `inst Graph`. I dette tilfellet er parameteren til `inst` en vanlig identifikator, `Graph`, som dermed angir navnet på templaten. Eller vi kan gjøre en instansiering med omnavning slik som i seksjon 4.2.2:

```
inst Graph(Node >> Junction(firstEdge >> firstRoad, \
                            lastEdge >> lastRoad, \
                            insertEdge(Node) >> insertRoad), \
           Edge >> Road(prevEdge >> prevRoad, \
                        nextEdge >> nextRoad))
```

Navnet på en template som skal instansieres angis med navnet på det ytterste metodekallet mens metodeparametrene angir eventuelle omnavninger. Dersom det ikke er noen omnavninger kan navnet på templaten angis med en enkel identifikator.

I dette eksemplet har `inst`-makroen et metodekall som parameter, og det er derfor navnet på metoden som angir navnet på templaten, mens de aktuelle parametrene til metodekallet angir en rekke omnavninger. Dette metodekallet tolkes således på en helt annen måte enn som et vanlig metodekall. Vi valgte denne metodekall-syntaksen til å spesifisere omnavninger siden en slik syntaks samsvarer ganske godt med den opprinnelige PT-syntaksen for omnavninger. Men betydningen er som sagt en helt annen.

Omnavningene angis som binære uttrykk med operatoren `>>`. Denne operatoren brukes altså i en annen betydning enn det som er den vanlige betydningen av `>>` i Boo: bitskifting mot høyre. Her brukes den kun for å skille gammelt og nytt navn fra hverandre. Over ser vi eksempel på en slik

omnavning: `Edge >> Road(prevEdge >> prevRoad, nextEdge >> nextRoad)`.

Venstre side i dette uttrykket er en identifikator som angir navnet på en klasse i templaten, og høyre side angir ett nytt navn for denne klassen. Høyre side kan enten rent syntaktisk være en identifikator eller et metodekall. Hvis høyre siden er en identifikator angir dette at det kun er klassen som skal gis nytt navn, men ingen av dens medlemmer. For eksempel om vi kun ønsket å gi nytt navn til klassen `Edge` men ikke til feltene `prevEdge` og `nextEdge` vil omnavningen se slik ut: `Edge >> Road` Hvis det siste er tilfellet at høyre side er et metodekall, slik det faktisk er i vårt eksempel, vil parametrene til dette kallet tolkes som omnavninger av metoder og felt i klassen. For omnavningen av `Edge` er det gitt to slike omnavninger, mens for omnavningen av `Node` er det gitt tre omnavninger: `firstEdge >> firstRoad`, `lastEdge >> lastRoad` og `insertEdge(Node) >> insertRoad)`.

Disse omnavningene må også angis som binære uttrykk med operatoren `>>`. Forskjellen er her at i dette tilfellet er det venstre side som enten kan være en enkel identifikator eller et metodekall, mens høyre side må være en identifikator. Høyre side angir det nye navnet for metoden eller feltet. Dersom venstre side er en identifikator, for eksempel som i `firstEdge >> firstRoad`, angir uttrykket en omnavning av et felt i template-klassen, men dersom venstre side er et metodekall, slik som `insertEdge(Node) >> insertRoad)`, angir dette omnavning av en metode. Et slikt kall blir imidlertid tolket som en metodesignatur isteden for et vanlig kall. En metode som gis nytt navn må nemlig angis med full signatur siden Boo tillater overlasting av metoder. De aktuelle parametrene i kallet er altså identifikatorer som tolkes som datatypene til parametrene til metoden som skal omnavnes.

På grunnlag av dette hierarkiet av metodekall og uttrykk i AST-et bygges det en tabell for omnavning av klasser og en todimensjonal tabell for omnavning av klassemedlemmene. Til dette benyttes Boo sin innebygde datastruktur for hash-tabeller. Den første tabellen av disse omnavningstabellene kaller jeg for enkelthets skyld *klassetabellen*. (*Klasseomnavningstabellen* ville være mer betegnende men også mer tungvint.) Denne indekseres med de opprinnelige klassenavnene, og peker til de nye navnene. Den andre tabellen, som jeg kaller *medlemstabellen*, indekseres med henholdsvis de opprinnelige klassenavne, og for hver av disse navnene på medlemmene i klassen. Sistnevnte er altså egentlig en tabell med pekere til andre hash-tabeller.

Initialisering

Etter bygging av disse omnavningstabellene utføres noe mer initialisering. Vi må blant annet opprette en tabell, om det ikke allerede er gjort i et tidligere kall til `inst`, over alle klasser som er instansiert i denne modulen. Denne tabellen kaller jeg for `inst`-*tabellen* og vil deles av alle kallene til `inst` i en

modul, og er nødvendig for å samordne sammenslåing av template-klasser. Dessuten må vi få en oversikt over alle `adds`-deklarasjoner.

Dette får vi gjennom `adds`-tabell som har blitt bygget under behandlingen av `adds`-attributtene. Siden attributter utføres før makrokall under kompileringen, kan vi være sikre på at denne tabellen er komplett. Disse to tabellene må være tilgjengelig for alle kall til `inst` i en modul. Derfor legges det til to annotasjoner på AST-noden til denne modulen med pekere til de to tabellene. Annotasjonen "inst" brukes for å referere til `inst`-tabellen. Denne tabellen vil senere fylles pekere til klassene i templaten etter hvert som de instansieres. Annotasjonen "adds" brukes for å referere til `adds`-tabellen. Denne siste tabellen er som nevnt allerede opprettet som følge av utførelsen `adds`-attributtene.

For å få tilgang til disse annotasjonene er det imidlertid nødvendig å finne modulen som inneholder makrokallet. Vi tillater at `inst` kun kan forekomme som en setning direkte i modulen der templaten skal instansieres. Alle slike setninger i en modul samles i en blokk som representeres med en egen node i AST-et, som altså vil ha modulen som foreldrenode. For å finne modulen som har dette makrokallet, må vi altså med utgangspunkt i makrokall-noden følge `ParentNode`-pekeren to ganger.

Finne templaten

Etter denne initialiseringen, forsøker `inst`-makroen å finne templaten med det gitte navnet. Dersom riktig .NET-assembly har blitt koblet inn i løpet av kompileringen, vil template-koden ligge i en klasse med samme navn som templaten. Template-navnet som blir gitt som en del av parameteren til `inst`-makroen er variabelt. Det tilsvarende klassenavnet kan derfor først fastslås når `inst`-makroen kjører, så dette kan ikke angis statisk i makrokoden. Derfor brukes det refleksjon[17] for å lage en instans av denne klassen, slik at vi kan lese ut feltet `_templateCode` som inneholder template-koden.

Intern kompilering av templaten

Når vi har fått tilgang til feltet med template-kode må koden kompileres. Det er nødvendig siden dette er en tekstlig kode, så den må kompileres til et AST som vi kan flette inn i AST-et til programmet som instansierer templaten. Dette gjøres ved å kalle en intern instans av Boo-kompilatoren. Dette blir på et vis altså andre gang templaten kompileres, siden templaten også ble kompilert i forbindelse med at den ble deklarert. Denne gangen bruker vi kompilatorsekvensen `ResolveExpressions`. Den er innebygget i Boo, og er en delsekvens av den som er standard. Den utfører alle steg fra parsing til og med navn- og typebinding. Dette gir oss et AST for template-klassene som

senere skal flettes inn i det instansierende programmet. Det viser seg at det er nyttig å la denne interne kompileringen sette opp navnebindinger for i dette treet.

Alle nodene i AST-et har en `Entity`-peker. Kompilatoren bruker denne `Entity`-pekeren til å angi navnebindinger for noder i AST-et som representerer bruksforekomster av type-, variabel- og metodenavn. For disse nodene angir `Entity`-pekeren en kobling mot nodene til de tilsvarende deklarasjonene. `Entity`-pekeren brukes også for å angi typebindinger, men for mange av nodetypene i AST-et settes ikke denne pekeren. Grunnen til at det er nyttig å sette opp disse navnebindingene er at det forenkler omnavning av metode- og feltdeklarasjoner. Det skal vi komme tilbake til. Disse bindingene må imidlertid etter hvert slettes, og opprettes på nytt etter at AST-et til template-klassene er flettet inn i AST-et til det instansierende programmet.

Utføre omnavninger

Etter at templaten er kompilert har vi tilgang til et AST med alle klassene i templaten. Disse skal flettes inn i AST-et til det instansierende programmet, men før det kan gjøres må omnavning av klasser, klassemedlemmer og referanser til disse utføres. Dette gjøres i henhold til omnavningstabellene som allerede har blitt bygget: klassetabellen og medlemstabellen. Omnavningene gjøres i følgende rekkefølge:

1 Omnavning av klassemedlemmer (metoder og felt)

2 Omnavning av klasser

3 Omnavning av typereferanser til klassene

4 Omnavning av referanser til klassemedlemmer (bruksforekomster)

Det er viktig i denne implementasjonen at omnavningene av deklarasjonene av klassemedlemmer gjøres først. Dette er fordi behandlingen av disse omnavningene er avhengig av å sammenligne metodesignaturene i en templateklasse med de som er gitt i spesifikasjonen av omnavningene. Dersom omnavning av typereferanser til klasser som har fått nye navn gjøres før dette, vil dette potensielt kunne endre metodesignaturene i template-klassene slik at en sammenligning med spesifikasjonene av omnavninger kan komme til å gi galt resultat.

Se for eksempel på instansieringen av templaten `Graph` i seksjon 4.2.2. Dersom omnavninger av typereferanser til klassen `Node` gjøres før omnavningen

av metoden insertEdge vil signaturen til insertEdge i template-klassen endres ettersom typen til parameteren endres fra Node til Junction. Den vil altså ikke lenger samsvare med omnavningen insertEdge(Node) >> insertRoad.

Etter omnavningen av klassemedlemmene gjøres altså omnavningene av klasser og typereferanser til klassene.

Deretter gjøres omnavningene av referanser til klassemedlemmene (bruksforekomster). Det er viktig at disse gjøres etter omnavningene av deklarasjonene av klassemedlemmene, siden omnavningene av bruksforekomstene baserer seg på at deklarasjonene allerede er gitt nye navn. Det som er viktig å passe på i forbindelse med omnavningen av slike bruksforekomster er at vi ikke endrer navnet på en lokal variabel som tilfeldigvis har samme navn som en av feltdeklarasjonene i en template-klasse. Dessuten kan for eksempel to forskjellige template-klasser ha en metode med samme navn. For å unngå dette utnytter vi at AST-et til template-klassene inneholder navnebindinger mellom bruksforekomster og metode- og feltdeklarasjoner. Som nevnt over finnes disse i Entity-pekeren til nodene i AST-et for disse bruksforekomstene. Siden omnavningene av deklarasjonene er gjort, kan vi enkelt gjøre en tilsvarende omnavning av bruksforekomstene ved å se på Entity-pekeren for hver bruksforekomst. For å gjøre dette har vi definert klassen RenameReferenceVisitor, som kan brukes til å opprette et visitor-objekt som kan brukes til å besøke alle bruksforekomster i templaten og sette navnet til hver bruksforekomst i henhold til Entity-pekeren.

Sletting av bindinger fra AST-et

Når omnavningene er gjort sletter vi alle bindinger fra AST-et med template-klassene. Dette gjøres ved å bruke en visitor-klasse RemoveBindingsVisitor som setter Entity-pekeren for alle noder i AST-et lik null. For uttrykk er det også nødvendig å sette nodens ExpressionType til null.

Dette må gjøres siden en template-klasse kan utvides med metoder som redefinerer ("override") metoder med samme navn i template-klassen. Et kall i en template-klasse til en slik redefinert metode må bindes til den nye metoden, mens de opprinnelige bindinger gjort da template-klassen ble kompilert vil peke til den opprinnelige metoden.

Bindingene vil bli gjenopprettet under senere steg i den regulære kompileringsprosessen, se seksjon 3.2.1.

Etter dette må template-klassene flettes inn i AST-et til det instansierende programmet. Hvordan dette gjøres for en gitt template-klasse, avhenger av om det allerede er instansiert en template-klasse med samme navn i programmet (i forbindelse med instansieringen av en annen template). Dette kan vi finne ut ved å slå opp i inst-tabellen.

Legge til utvidelser (adds-klasser)

Dersom det for en template-klasse ikke finnes en klasse med samme navn i `inst`-tabellen, må template-klassen legges direkte inn i det instansierende programmet. Først må imidlertid eventuelle utvidelser gitt i en `adds`-klasse legges inn i template-klassen. Denne `adds`-klassen finner vi ved å slå opp i `adds`-tabellen, som allerede er opprettet i forbindelse med behandlingen av `adds`-attributtene. Deretter legges alle medlemmene i `adds`-klassen inn i template-klassen

Dersom en metode i `adds`-klassen redefinerer en metode i template-klassen må denne behandles spesielt.

Anta for eksempel at en metode `m2` i `adds`-klassen redefinerer en metode `m1` med samme signatur i template-klassen, og at de begge har navnet `m`. Da må vi gi den eksisterende metoden `m1` et nytt navn som ikke kolliderer med andre navn. Vi kan ikke bare forkaste en metode som er redefinert, siden vi skal tillate at metoder i `adds`-klassen kan referere til den opprinnelige metoden `m1` ved å bruke `tsuper`-mekanismen.

Vi bruker en enkel algoritme for omnavning av den opprinnelige metoden `m1`. Dette navnet vil ikke være synlig for brukere av PT-mekanismen, som isteden som sagt kan kalle den opprinnelige metoden fra metoder i `adds`-klassen ved å bruke `tsuper.m(...)`. Det er altså vilkårlig hvilket navn vi velger for `m1` så lenge det ikke kolliderer med andre navn.

Vi har allerede nevnt i seksjon 3.2.2 at Boo-parseren ikke tillater identifikatorer med dollartegn, men at vi kan opprette slike i resten av kompilatorsekvensen. Vi velger derfor et nytt navn som er det samme som det opprinnelige med et dollartegn tilføyd til slutt: `m$`.

Det er verdt å legge merke til en ting i denne sammenhengen. Templater kan instansieres i templater slik det ble bemerket i seksjon 2.7. Det betyr at en metode kan redefineres flere ganger. Det er altså mulig at den opprinnelige utgaven av metoden, `m1`, selv har redefinert en metode `m0`, som har hatt det samme navnet, altså `m`.

Det betyr i forbindelse med instansieringen av templaten der `m0` er definert så har `m0` fått nytt navn: `m$`. Det kan derfor tilsynelatende oppstå en konflikt om også `m1` gis dette navnet. Dette problemet vil imidlertid ikke oppstå i denne implementasjonen av PT, siden hver gang en template kompileres blir alle identifikatorer som inneholder dollartegn oversatt til gyldige identifikatorer. Dette ble forklart i seksjon 4.3.1. Dette betyr at når `m1` gis navnet `m$` så vil `m0` egentlig ha navnet `m_`, eventuelt med et suffiks, og som altså ikke gir noen konflikt.

Etter at redefinerte metoder er omnavnet vil alle `tsuper`-kall som forekommer i `adds`-klassen erstattes med kall som har de faktiske metodenavnene.

Endelig, etter at alle metodene fra `adds` er flettet inn, kan template-klassen settes inn i det instansierende programmet, og til slutt registrer vi at dette er gjort ved å legge inn en peker til AST-noden for denne klassen i `inst`-tabellen.

Sammenslåing av instansierte klasser med samme navn

Dersom det allerede finnes en klasse med samme navn i `inst`-tabellen for en template-klasse, må template-klassen ikke legges direkte inn i det instansierende programmet. Dette er fordi klassen i `inst`-tabellen med det samme navnet allerede er lagt inn i AST-et til det instansierende programmet. De to klassene med samme navn skal isteden slås sammen slik som angitt i seksjon 2.5. Medlemmene i template-klassen må altså flettes inn i klassen fra `inst`-tabellen. Fremgangsmåten for dette er relativt rett frem. Vi må imidlertid kontrollere at det ikke forekommer navnekollisjoner mellom medlemmene i de to klassene.

Dessuten må vi sjekke om det er metoder fra `adds`-klassen som redefinerer noen av metodene som flettes inn. Hvis dette er tilfelle må metodene som flettes inn gis nye navn på samme måte som forklart over.

4.4 Avsluttende vurderinger

Vi har nå sett ganske detaljert på hvordan BooPT er implementert ved hjelp av Boo sine makroer og syntaktiske attributter. Dette viser at det altså er mulig å lage en enkel variant av PT ved hjelp av Boo sin funksjonalitet for metaprogrammering. Når en regner antall kodelinjer (under 1000) er implementasjonen er ikke omfattende.

Metaprogrammeringsmekanismene i Boo og den fleksible kompilatorarkitekturen har i stor grad lettet arbeidet og gitt en kompakt implementasjon. Et poeng med denne implementasjonen er at det ikke har vært nødvendig å modifisere Boo-kompilatoren for å få den til å virke og det har heller ikke vært nødvendig å bruke en annen kompilatorsekvens enn den som er standard. Dette gir i seg selv en indikasjon på styrken til metaprogrammeringsmekanismene i Boo. Det gjør også at det er enkelt for en bruker/programmerer som ønsker å benytte seg av BooPT ettersom Boo-kompilatoren kan kjøres på helt vanlig måte. Det eneste å passe på er at navnerommet `BooPT` er tilgjengelig under kompileringen.

Vi skal nå se på andre sider ved implementasjonen som belyser styrke og svakheter ved Boo sin funksjonalitet for metaprogrammering.

4.4.1 Syntaks

Syntaksen vi har brukt for de forskjellige konstruksjonene i BooPT baserer seg på syntaksen Boo tilbyr for makrokall og bruk av syntaktiske attributter. Boo har for øyeblikket ikke støtte for å utvide syntaksen i språket utover det som er mulig ved hjelp av disse mekanismene. Disse har likevel gitt oss en ganske grei syntaks som ligger nokså tett opptil den syntaksen for som er beskrevet i kapittel 2, selv om den naturlig nok er mer tilpasset vanlig Boo-syntaks og har ikke de Java-liknende elementene som er brukt i kapittel 2.

Vi har for eksempel oppnådd en grei syntaks for setningene `template` og `inst` ved bare å bruke vanlige makrokall. Syntaksen for å angi `adds`-klasser med attributter skiller seg litt mer fra den opprinnelige PT-syntaksen, men det er antagelig den beste tilnærmingen som er mulig å få til i Boo.

Da vi gikk igjennom implementasjonen i 4.3 så vi dessuten hvordan det er mulig å bruke metaprogrammeringsmekanismene i Boo til å endre betydningen av vanlig Boo-syntaks. Dette har vært viktig for implementasjonen BooPT. Vi har for eksempel sett at en kombinasjon av metodekall og bruk av operatoren `>>` tolkes som noe helt annet når de gis som parameter til `inst`-makroen.

Dessuten har vi sett at det er mulig å gi ordet `tsuper`, som i utgangspunktet bare er en vanlig identifikator i Boo, en helt spesiell betydning når det forekommer i `adds`-klasser.

Selv om disse mekanismene har gitt en grei syntaks for PT, så kunne det vært ønskelig at syntaksen var litt klarere på enkelte punkter, samt at den var enda mer i overensstemmelse med den opprinnelige PT-syntaksen. Jeg vil nå gå igjennom noen av disse punktene.

Template-deklarasjonen i BooPT angir at alle klassene i modulen der deklarasjonen forekommer skal være med i templaten. Det ville imidlertid være mer i overensstemmelse med vanlig PT-syntaks, og kanskje være mer naturlig, om en slik deklarasjon kunne gjøres gjeldende for klassene i en etterfølgende blokk. Koden under viser et eksempel på en template-deklarasjon slik vi altså helst kunne ønske oss den:

```
import BooPT

template Graph:
  class Node:
    firstEdge as Node = null
    lastEdge as Node = null

    def insertEdge(to as Node) as Edge:
      ...

  class Edge:
```

```
from as Node
to as Node
prevEdge as Edge = null
nextEdge as Edge = null

def constructor(from as Node, to as Node):
    ...
```

Hvis vi sammenligner med figur 4.1, så ser vi at forskjellen er at klassene i templaten her angitt i en egen blokk, etter template-deklarasjonen. Dette ligner jo også på hvordan innholdet i klasser angis i Boo. Med en slik syntaks ville det dessuten være mulig å definere flere templater i en Boo-modul.

En slik løsning er imidlertid dessverre ikke mulig med gjeldende versjon av Boo, siden Boo ikke tillater at kroppen til en makro inneholder klasser, verken en eller flere. Det er likevel interessant at Boo støtter makrokropper med opptil flere metodedefinisjoner, så vi kan kanskje håpe at en fremtidig versjon av Boo også vil gi denne muligheten for klasser.

Som nevnt har Boo ikke støtte for utvide syntaksen i språket utover det som er mulig med makroer og syntaktiske attributter. Det er blant annet ikke mulig med egendefinerte operatorer eller overlasting av eksisterende operatorer. For BooPT kunne det for eksempel være ønskelig å opprette nye operatorer til bruk i omnavningene. BooPT bruker `>>` for å skille gammelt og nytt navn fra hverandre i en omnavning, men vanlig PT-syntaks bruker `=>` og `->` som ser litt bedre ut. Støtte for utvidbar syntaks er imidlertid under utvikling, og en prototype kalt OMeta[18] har vært tilgjengelig en stund for utprøving men denne har ikke ennå blitt en fullverdig del av Boo.

Til slutt en betraktning om syntaksen for instansiering i BooPT. Den opprinnelige Java-liknende PT-syntaksen spesifiserer at instansiering angis med en `inst`-setning. En `inst`-setning kan imidlertid bli ganske lang dersom det er mange omnavninger, men med en Java-liknende syntaks er det uproblematisk å dele denne setningen over flere linjer. BooPT sin `inst` etterligner dette så langt det er mulig, og utgjør også en enkel setning. Om det er mange omnavninger, er det også mulig å dele denne setningen over flere linjer, men siden linjeskift i Boo angir slutten av en setning, er det nødvendig å bruke et spesielt symbol for å indikere at en setning fortsetter på neste linje. Vi har sett eksempel på dette tidligere:

```
import BooPT

inst Graph(Node >> Junction(firstEdge >> firstRoad, \
                            lastEdge >> lastRoad, \
                            insertEdge(Node) >> insertRoad), \
           Edge >> Road(prevEdge >> prevRoad, \
                        nextEdge >> nextRoad))
```

I BooPT ville det derfor kanskje vært mer naturlig å bruke en egen blokk

for å angi omnavninger. For eksempel slik:

```
inst Graph:
  with Node >> Junction(firstEdge >> firstRoad, lastEdge >> lastRoad, \
                        insertEdge(Node) >> insertRoad)
  with Edge >> Road(prevEdge >> prevRoad, nextEdge >> nextRoad)
```

Eller eventuelt slik:

```
inst Graph:
  with Node >> Junction:
    with firstEdge >> firstRoad
    with lastEdge >> lastRoad
    with insertEdge(Node) >> insertRoad

  with Edge >> Road:
    with prevEdge >> prevRoad
    with nextEdge >> nextRoad
```

Dette er et eksempel på nøsting av makroer og ville kreve en ny makro `with`. For øvrig blir `with` å betrakte som "syntaktisk sukker" som gjør implementasjonen av denne syntaksen noe mer komplisert. Som en variant av det første forslaget, kan man for eksempel utelate `with` men ellers beholde den samme blokk-strukturen.

4.4.2 Format for kompilert template

En sentral utfordring i denne oppgaven har vært å finne et passende format for å gjøre en kompilert template tilgjengelig for senere instansiering i et program. Vi har valgt å bruke et .NET-assembly til dette formålet, men innholdet er litt spesielt i den forstand at assembliet ikke inneholder kompilert bytekode for templaten. Isteden inneholder det en tekstlig Boo-kode som svarer til definisjonen av templaten.

Dette kan i utgangspunktet oppfattes som en spesiell løsning. Grunnen til at en slik løsning er at valgt er at dette gir en måte å utsette den endelige kompileringen av templaten til templaten skal instansieres. Dette er et av hovedpoengene med PT slik det er beskrevet i kapittel 2. Når templaten instansieres vil den altså kompileres til et AST, og dette vil flettes sammen med AST-et for det instansierende programmet.

Det kan kanskje betraktes som en nokså lite elegant løsning. For det første krever den for eksempel at templaten kjøres gjennom kompilatoren mer enn en gang både når den deklareres og instansieres. Vi har sett at både `template`-makroen og `inst`-makroen kaller Boo-kompilatoren internt for å omforme templaten til et format som passer for videre behandling. Dette kan regnes som lite effektivt. En annen ting er at denne løsningen bruker et .NET-assembly til noe det ikke egentlig er beregnet for. Et .NET-assembly er et filformat beregnet for å lagre bytekode ikke tekstlig kildekode.

Vi kunne tenke oss andre mulige løsninger, men disse har også sine ulemper. Vi kan for eksempel spørre oss om ikke `inst` like gjerne kunne hente en template direkte fra den opprinnelige filen med kildekode. Det kunne den gjøre og det hadde vært et alternativ. Da ville det ikke være nødvendig å kompilere templaten før instansiering i et program. Det er imidlertid noen fordeler ved å kompilere templaten først:

For det første, ved å kompilere templaten for seg forsikrer vi oss om at den er fri for kompileringsfeil før den kan tas i tatt bruk. Dersom instansieringen av en template feiler må det altså skyldes noe annet enn en feil i templaten, for eksempel navnekollisjoner. Dette sikrer oss selvsagt ikke mot logiske feil i templaten som ikke kan oppdages under kompileringen.

For det andre, tilbyr Boo en standardisert metode for å angi koblinger til .NET-assembly som et program er avhengig av i forbindelse med kjøring eller kompilering av programmet. Det finnes ingen tilsvarende mekanismer for å angi koblinger til filer med kildekode. Dette er naturlig siden Boo ikke er et språk som blir tolket.

Dessuten vil en kompilert template også inkludere andre templater som instansieres i denne templaten. En template kompilert til et .NET-assembly kan altså inneholde kode fra mange templater. Om vi ved instansiering derimot baserer oss på å kompilere filen med kildekode for templaten, er det også nødvendig å koble inn og kompilere kildekoden for templatene som denne selv instansierer, og så videre. Dette kan forårsake at bruk av templater oppfattes som unødvendig tungvint.

En annen mulighet ville være å finne et helt annet format for lagring av templaten etter den er kompilert. Et slikt mulig format er XML-representasjonen til et Boo-program, slik som for eksempel vist i figur 3.4. Dette ser ut til å være egnet siden det gir en komplett representasjon av AST-et til et program. Dessuten tilbyr Boo som vi har sett metoder for å serialisere AST-et til en XML-representasjon. En grunnen til at en slik løsning ikke ble valgt er at Boo for øyeblikket dessverre ikke har noen støtte for å deserialisere et slikt XML-tre tilbake til et AST.

En siste forslag til et slikt format vil være et element i det som i [4] kalles for en homogen implementasjon av PT. Som det antydes i [4] vil en slik implementasjon være svært forskjellig fra en type av PT som er beskrevet i kapittel 2, inkludert BooPT. Tanken med en slik løsning er at det kun er én kopi av templaten for et kjørende program, selv om denne instansieres flere ganger, og instansieringer, omnavninger og utvidelser må håndteres ved å bruke en form for tabellverk, samt andre mekanismer. Det er imidlertid ennå ikke laget en slik implementasjon av PT og, hvordan en slik løsning vil fungere er heller ikke fullstendig avklart ennå. Men det virker rimelig å anta at med en slik implementasjon i Boo vil man kunne kompilere en template til bytekode i et vanlig .NET-assembly. Denne løsningen er imidlertid ganske

mye på siden av det som er realistisk for denne oppgaven.

4.4.3 Organisering av implementasjonen

Koden til BooPT er organisert på en forholdsvis oversiktlig måte. For hver av de sentrale konstruksjonene i PT, `template`, `inst` og `adds`, er det definert en makro eller et syntaktisk attributt, og disse, inkludert noen hjelpeklasser, utgjør all funksjonalitet som kreves i denne PT-implementasjonen.

I tillegg til at denne løsningen gir en grei inndeling av koden på et overordnet nivå, så har den en fordel ved at den ikke krever noen inngrep i kompileringsprosessen utover det som gjøres av makroene og attributtet. Vi trenger for eksempel ikke å modifisere Boo-kompilatoren på noen måte, og vi trenger heller ikke å bruke en egendefinert kompilatorsekvens. Vi kan bruke den vanlige kompilatorsekvensen til Boo-kompilatoren.

Det er imidlertid også en ulempe med denne løsningen. Dette gjelder spesielt implementasjonen av `inst`-makroen. Denne makroen håndterer alt som har med instansieringen av en template å gjøre med en gang den kalles. Det ville imidlertid vært ryddigere å gjøre instansieringene i to steg:

1 Registrere alle instansieringer inkludert omnavninger

2 Utføre alle instansieringene

Hovedgrunnen til dette er at template-klasser fra forskjellige instansieringer kan slås sammen dersom de gis samme navn. Det kan derfor være en fordel å vente med å flette template-klassen fra en instansiering sammen med en eventuell `adds`-klasse og inn i det instansierende programmet før vi vet om den skal slås sammen med andre template-klasser.

Slik `inst`-makroen er implementert er dette ikke mulig. Når et kall til `inst` oppdager at en template-klasse skal slås sammen med en template-klasse som allerede er instansiert i et tidligere kall `inst` vil den sistnevnte template-klassen allerede være flettet inn i det instansierende programmet. Dette gjør det blant annet spesielt utfordrende når slike template-klasser har `adds`-klasser med metoder som redefinerer metoder i template-klassene. Dette håndteres i BooPT men ikke på en spesielt elegant måte, og det vil trolig være enda mer utfordrende for en mer omfattende implementasjon av PT.

En løsning på dette vil være å finne en mekanisme som gjør det mulig å gripe inn i kompilatoren etter at alle kall til `inst` er utført.

En kunne tenke seg å gjøre dette i det siste kallet til `inst` i en modul. Men et kall til `inst` har ingen god måte å avgjøre om dette er det siste i den gjeldende modulen. Alle kall til `inst` kan selvsagt inspisere hele AST-et

for å undersøke om det er senere kall. Det heller ikke spesielt elegant, men kanskje likevel bedre enn den som nå er valgt.

Jeg vil også foreslå en annen løsning, men den vil innebære en utvidelse av Boo-kompilatoren, så det er ingen realistisk løsning på kort sikt. Vi har sett at makroer og de andre metaprogrammeringsfasilitetene kan legge inn annotasjoner med vilkårlige verdier i nodene AST-et. Det kunne kanskje være nyttig med en tilsvarende mekanisme for å annotere noder med kode som skal utføres på et senere steg i kompileringen. Dette kan betraktes som en type *hook*-kode[19].

Med en slik mulighet kunne `inst`-makroen sette inn en slik hook for noden til en Boo-modul som utførte alle instansieringer etter de har blitt registrert av kall til en `inst`-makroen.

En kunne tenke seg at slike annotasjoner ble utført i et eget, nytt steg i kompilatorsekvensen, som i hvertfall måtte settes inn etter steget som ekspanderer makroer og syntaktiske attributter, `MacroAndAttributeExpansion`. Eller en kunne tenke seg en mer avansert mulighet der en kan angi hvor i kompilatorsekvensen en bestemt hook skal utføres, for eksempel før eller etter et bestemt kompilatorsteg.

En fordel med en slik utvidelse av Boo er at den for eksempel ikke krever noen utvidelse av syntaksen til Boo, og burde derfor ganske enkelt la seg implementere i Boo. Ulempen med en slik mulighet er at den kanskje kan gi for stor fleksibilitet og dermed også uforutsigbarhet om hva som faktisk skjer i løpet av kompileringen. For eksempel om flere forskjellige makroer skulle opprette slike hooks uten å samarbeide er det lett å tenke seg at det kan oppstå feil.

Den antageligvis mest opplagte løsningen er å introdusere et egendefinert kompilatorsteg som behandlet og utførte alle instansieringer etter at alle disse har blitt registrert av kall til en `inst`-makro. Ulempen med en slik løsning at dette i prinsippet innebærer en utvidelse av Boo-kompilatoren, og det krever at programmerere som skal bruke BooPT må være klar over at kompilatoren må kjøres med en spesiell kompilatorsekvens, se seksjon 3.2.4.

Kapittel 5

Status for BooPT og videre arbeid

Vi har sett er det er mulig å få til en enkel implementasjon av PT ved å benytte metaprogrammeringsegenskapene til Boo. Disse egenskapene har bidratt til at arbeidet med implementasjonen har vært overkommelig innenfor den gitte tidsrammen.

Uten disse mekanismene måtte vi for eksempel programmere støtte både for å gjenkjenne (parse) den spesielle syntaksen i PT, og for å generere Boo-koden som blir resultatet av instansieringer. Boo sin støtte for makroer og syntaktiske attributter gjør at vi kan bruke Boo-kompilatoren sin innebygde parser, samt utnytte mulighetene for å manipulere AST-et.

Det har likevel vært en del utfordringer knyttet til implementasjonen slik vi har vært inne på i seksjon 4.4, blant annet for å finne et passende format for å gjøre en kompilert template tilgjengelig for senere instansiering i et program. Dessuten har det også vært utfordringer knyttet til både det å sette seg inn i de avanserte mulighetene til språket Boo og selve Boo-kompilatoren, blant annet på grunn av at det er begrenset med dokumentasjon på dette området.

Det er noe arbeid som gjenstår før vi kan si at vi har en fullgod implementasjon av PT for Boo, men for mye av dette kan jeg ikke se noen store hindringer, slik at dette greit skulle la seg gjennomføre gitt noe mer tid. Nedenfor gis noen forslag til videre arbeid på dette området.

5.1 PT-egenskaper som ikke er implementert

Arbeidet med BooPT har tatt sikte på å støtte alle de grunnleggende egenskapene ved PT-mekanismen som er beskrevet i kapittel 2, riktignok med unntak av typeparametrisering (seksjon 2.6). Dette har stort sett vært vel-

lykket, men på grunn av begrensinger i tid er det enkelte aspekter ved PT som ennå ikke har blitt implementert i BooPT.

BooPT støtter blant annet foreløpig ikke bruk av `tsuper` for å kalle den opprinnelig konstruktøren i en template-klasse fra en konstruktør angitt i `adds`-klassen. Dette er som nevnt i seksjon 2.5 nyttig i forbindelse med sammenslåing av klasser. Gjeldende implementasjon av BooPT håndterer ikke eventuelle konstruktører i instansierte klasser, men forutsetter at `adds`-klassen angir en ny konstruktør. Implementasjonen støtter imidlertid `tsuper` for å kalle vanlige metoder som har blitt redefinert i `adds`-klasser, og det antas at det ikke vil være vanskelig å legge til støtte `tsuper` for konstruktører ved å ta utgangspunkt i hvordan `tsuper` blir håndtert for vanlige metoder. Det kan imidlertid være en utfordring knyttet til tilordning av felter som er deklarert som `final`, det vil si at det kun kan tilordnes verdi i konstruktøren.

I tillegg til metoder og felt har Boo en tredje type klassemedlemmer, såkalte egenskaper. Et eksempel ble gitt i begynnelsen av kapittel 3. Som en forenkling har BooPT kun støtte for metoder og felt i klassene i en template, men det er klart at en fullgod implementasjon av BooPT også bør støtte slike egenskaper. Dette bør ikke være vanskelig å få til ved å ta utgangspunkt i den eksisterende funksjonaliteten for å håndtere metoder. Egenskaper kan i likhet med metoder redefineres, så funksjonaliteten som er nødvendig for å håndtere slike vil være svært lik bare enklere ettersom egenskaper ikke har parametre.

Det burde også være overkommelig å implementere mer avanserte egenskaper ved PT slik som typeparametrisering, selv om dette nok vil kreve noe mer arbeid.

5.2 Ekstra kompilatorsteg

Dersom BooPT-implementasjonen utvides, for eksempel med støtte for de mer avanserte egenskapene til PT bør det vurderes om implementasjonen bør organiseres på en annen måte slik det ble diskutert i seksjon 4.4.3. Det vil antagelig gi større fleksibilitet og lette implementasjon av nye egenskaper dersom selve koden som utfører instansieringene (punkt 2 i seksjon 4.4.3) gjøres i et eget kompilatorsteg.

5.3 Syntaks

Vi har, blant annet i seksjon 4.4.1, vurdert hvordan makroer og syntaktiske attributter egner seg til å konstruere en syntaks for PT. Vi fant at disse har visse begrensninger, men for en enkel variant av PT slik som BooPT

gir de likevel en grei syntaks. Det ble nevnt at en prototype OMeta[18] for mer omfattende støtte for utvidbar syntaks i Boo er under utvikling. Et forslag til videre arbeid er å utforske de mulighetene OMeta gir for å utvide syntaksen i Boo, og om dette eventuelt kan gi en syntaks som ligger tettere opp mot den opprinnelige PT-syntaksen beskrevet i kapittel 2. Det kan dessuten være nyttig med større fleksibilitet til å definere ny syntaks med tanke på å støtte typeparametrisering av templater. Til dette kreves en nokså omfattende syntaks, se [4].

5.4 Testing

Implementasjonen har ikke vært gjenstand for omfattende og fullstendig uttesting, så det må tas høyde for at det kan forekomme feil. Disse burde det imidlertid være overkommelige å rette. Under arbeidet med implementasjonen og i ettertid har det av og til dukket opp feil, men disse har som regel vært greie å løse. Det er utarbeidet en del tester for å kontrollere at de forskjellige aspektene ved implementasjonen fungerer som de skal, men det ville være ønskelig å utarbeide et mer omfattende og systematisk sett med enhetstester.

5.5 Konklusjon

Under arbeidet med denne oppgaven er det laget en enkel implementasjon av PT-mekanismen for Boo og vi har sett at Boo sine makroer og syntaktiske attributter i all hovedsak er velegnet for en slik implementasjon.

Det har imidlertid vært enkelte utfordringer knyttet til implementasjonen av BooPT. Det er også rom for videre forbedring av BooPT, og således muligheter for videre arbeid.

Konklusjonen blir at til tross for enkelte utfordringer så er Boo sine mekanismer for metaprogrammering er godt egnet til oppgaven, og trolig er disse mekanismene også godt egnet til å lage en mer omfattende og komplett variant av PT.

Bibliografi

[1] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen, "Exploring the use of package templates for flexible re-use of collections of related classes," *Journal of Object Technology*, vol. 8, no. 7, pp. 59–85, 2009.

[2] R. B. de Oliveira, "The boo programming language," 2005, [besøkt 2011-12-19]. [Online]. Available: http://boo.codehaus.org/BooManifesto.pdf

[3] "Common language infrastructure (cli)," ECMA-335, ECMA International, Dec. 2010. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[4] S. K. Eyvind W. Axelsen, Fredrik Sørensen and B. Møller-Pedersen, "Challenges in the design of the package template mechanism," *Transactions on Aspect-Oriented Software Development*.

[5] "Boo website," [besøkt 2011-12-19]. [Online]. Available: http://boo.codehaus.org/

[6] C. K. Knight, "Boo primer," [besøkt 2011-12-19]. [Online]. Available: http://boo.codehaus.org/Boo+Primer

[7] "Boo language guide," [besøkt 2011-12-19]. [Online]. Available: http://boo.codehaus.org/Language+Guide

[8] A. Rahien, *DSLs in Boo: Domain Specific Languages in .NET*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[9] "Boo source code repository," [besøkt 2011-12-19]. [Online]. Available: https://github.com/bamboo/boo/

[10] "Boo syntax diagram," [besøkt 2011-12-19]. [Online]. Available: http://boo.codehaus.org/syntax-diagram.html

[11] "Boo ast-modell," [besøkt 2011-12-19]. [Online]. Available: https://github.com/bamboo/boo/blob/master/ast.model.boo

[12] "Boo api-dokumentasjon (tilgjengelig fra en tredjepart)," [besøkt 2011-12-19]. [Online]. Available: http://boo.sourcearchive.com/

[13] "Boo version 0.9.4," Jan. 2011, [besøkt 2011-12-19]. [Online]. Available: http://dist.codehaus.org/boo/distributions/

[14] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers, May 2007.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] M. Corporation, *Microsoft Visual C# .NET Language Reference.* Redmond, WA, USA: Microsoft Press, 2002.

[17] B. C. Smith, "Procedural reflection in programming languages," Ph.D. dissertation, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982. [Online]. Available: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-272.pdf

[18] "Boo ometa," [besøkt 2011-12-19]. [Online]. Available: https://github.com/bamboo/boo-extensions/

[19] S. Black, "Design pattern: Hook operations," [besøkt 2011-12-19]. [Online]. Available: http://stevenblack.com/PTN-HookOperations.html

Tillegg A

Kildekode for BooPT

Dette tillegget inneholder kildekoden til BooPT-implementasjonen. Koden inneholder en del kommentarer. Disse er på engelsk, noe som også gjelder alle identifikatorer og navn. Det er rom for forbedringer av koden. Det er blant annet en del kommentarer som starter med ordet FIXME. Disse indikerer ikke nødvendigvis at det er noe direkte galt med koden, men at det er rom for forbedringer eller ønskelig å legge til utvidet funksjonalitet. Koden er for øvrig utførlig forklart i seksjon 4.3 i denne oppgaven.

A.1 `TemplateMacro`

```
namespace BooPT

import System
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast
import Boo.Lang.Compiler.IO


class TemplateMacro(AbstractAstMacro):

  override def Expand(macro as MacroStatement):

    if not (macro.Arguments.Count == 1 \
      and macro.Arguments[0] isa ReferenceExpression):
      raise "template name missing or invalid"
    templateName = (macro.Arguments[0] as ReferenceExpression).Name

    if not (macro.ParentNode.ParentNode isa Module):
      raise "template can only be declared at module level: " + templateName

    /* FIXME: We should check that the template macro is only called once
       in a module, or things could get ugly. */
```

```
parentModule as Module = macro.ParentNode.ParentNode
parentModule.Globals.Statements.Remove(macro)

/* Check that the template only contains class definitions. FIXME:
   Interfaces should also be allowed. Perhaps also enum and
   callable definitions could be allowed, although these
   constructs are not covered by the original specification of PT
   so such constructs would be specific to Boo. */
for member in parentModule.Members:
  if not (member isa ClassDefinition):
    raise "template can only contain class definitions: " + templateName

/* Compile all members of the template without generating any code.
   This utilizes a custom pipeline with a special step that translates
   identifiers with dollar signs in them to legal identifiers. */
compiler = BooCompiler()
compiler.Parameters.Pipeline = CompileWithTranslateGeneratedNamesStep()
compiler.Parameters.Input.Add(StringInput("code", parentModule.
    ToCodeString()))
/* Make sure to reference the same libraries as in the current compiler
   context. This is necessary when templates have references to external
   libraries. */
compiler.Parameters.References = CompilerContext.Current.Parameters.
    References
context = compiler.Run()

/* Errors are not reported unless we explicitly ask for it. */
if (len(context.Errors) > 0):
  raise "template compilation failed:\n" + context.Errors

templateModule as Module = context.CompileUnit.Modules[0]

/* Create a field to store code of this template. */
codeValue = StringLiteralExpression(Value: templateModule.ToCodeString())
codeField = Field(Name: "_templateCode", Initializer: codeValue)

/* Create a new class with a field containing the template code
   and add it to the now empty module. */
templateCodeClass = ClassDefinition()
templateCodeClass.Name = templateName

/* Add the field to the template. */
templateCodeClass.Members.Add(codeField)

/* Remove all members, including classes and other type definitions,
   and global statements from the module. The result will be a
   completely empty module until the class containing the literal
   template code are added. */
parentModule.Members = TypeMemberCollection()
parentModule.Globals = Block()

/* Add the class with template code. */
parentModule.Members.Add(templateCodeClass)
```

A.2 `InstMacro.boo`

```
namespace BooPT

import System
import System.Text.RegularExpressions
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Pipelines
import Boo.Lang.Compiler.IO
import Boo.Lang.Compiler.Ast


/* Instantiate a template with zero or more renamings. Invocations
   must have the following form:

   inst TemplateName(C1 >> D1, C2 >> D2(M1 >> P1, ...), ...)

   where Cn are exisiting template class names, Dn are the new
   class names, Mn are names of members in exisiting template classes
   and Pn are the new member names. */
class InstMacro(AbstractAstMacro):
  /******************* Field declarations ***************/

  /* Template name */
  private templateName as string

  /* Declare and initialize hashes to contain class and member
     renamings respectively. FIXME: Should probably use a more
     suitable data structure. */
  private classRenaming as Hash = {}
  private memberRenaming as Hash = {}


  /******************** Helper methods *****************/

  /* Find and return the specified type among the
     loaded assemblies. */
  private static def FindType(typeStr as string) as Type:
    for assembly in AppDomain.CurrentDomain.GetAssemblies():
      type = assembly.GetType(typeStr)
      if not (type is null):
        return type
    return null


  /* Check whether two type references are equal. */
  private static def MatchType(type1 as TypeReference, \
                               type2 as TypeReference) as bool:
    simpType1 as SimpleTypeReference = type1
    simpType2 as SimpleTypeReference = type2
    return (simpType1.Name == simpType2.Name)
```

```
/* Check whether two parameter collections are equal. */
private static def MatchParameters( \
  parameters1 as ParameterDeclarationCollection, \
  parameters2 as ParameterDeclarationCollection) as bool:

  if parameters1.Count != parameters2.Count:
    return false
  for i in range(0, parameters1.Count):
    param1 as ParameterDeclaration = parameters1[i]
    param2 as ParameterDeclaration = parameters2[i]
    if not MatchType(param1.Type, param2.Type):
      return false
  return true


/* Check whether a collection of reference expressions matches a
   parameter collection. */
private static def MatchParameters(arguments as ExpressionCollection, \
   parameters as ParameterDeclarationCollection) as bool:

  if arguments.Count != parameters.Count:
    return false
  for i in range(0, arguments.Count):
    if not (arguments[i] isa ReferenceExpression):
      return false

    arg as ReferenceExpression = arguments[i]
    param as ParameterDeclaration = parameters[i]
    argTypeName = arg.ToString()
    paramType as SimpleTypeReference = param.Type

    if argTypeName != paramType.Name:
      return false
  return true


/* Check whether two class members match, that is if they
   have the same name and, in the case of methods, signatures. */
private static def MatchMembers(member1 as TypeMember, \
                                member2 as TypeMember) as bool:
  if member1.Name != member2.Name:
    return false

  if member1 isa Field and member2 isa Field:
    raise "fields cannot be overriden: $(member1.FullName)"

  if member1 isa Method and member2 isa Field:
    raise "cannot override method with a field: $(member1.FullName)"

  if member1 isa Field and member2 isa Method:
    raise "cannot override field with a method: $(member1.FullName)"

  if member1 isa Method and member2 isa Method:
    memberMethod1 as Method = member1
```

```
    memberMethod2 as Method = member2
    if not MatchParameters(memberMethod1.Parameters, \
                           memberMethod2.Parameters):
      return false

  /* FIXME: Return type checking does not yet work. */
  # if not MatchType(memberMethod1.ReturnType, \
  #                  memberMethod2.ReturnType):
  #   raise "cannot override $(memberMethod1.FullName)" \
  #     + "with a method of a different return type: " \
  #     + memberMethod2.ReturnType

  return true


/* Find the real method name corresponding to a tsuper call. That
   is, we want the method with the most dollar signs appended to the
   given name. This method is also used by the class ResolveTSuperVisitor
   and is therefore not private.
   FIXME: We probably only need to check for a single dollar sign suffix
   since templates are compiler with the TranslateGeneratedNamesStep,
   so dollar signs are translated into underscores. */
internal static def FindTSuperMethodName(classDef as ClassDefinition, \
                                         name as string) as string:
  returnName = name
  regexp = Regex("^" + name + "\\$*$")
  for memb in classDef.Members:
    if memb isa Method and regexp.IsMatch(memb.Name) \
      and len(memb.Name) > len(returnName):
      returnName = memb.Name
  return returnName


/* Process the macro arguments. That is, read the template name, and
   the renaming declarations of the inst statement and translate
   them into a more appropritate datastructure. */
private def ReadArguments(args as ExpressionCollection):
  /* Read and interpret the macro argument. The template name
     and none or more renamings must be specified in a single
     macro argument. */
  if len(args) != 1:
    raise "template name required"
  elif args[0] isa ReferenceExpression:
    templateName = cast(ReferenceExpression,  args[0]).Name
  elif args[0] isa MethodInvocationExpression \
    and (args[0] as MethodInvocationExpression).Target \
    isa ReferenceExpression:

    classRen = cast(MethodInvocationExpression, args[0])
    templateName = cast(ReferenceExpression, classRen.Target).Name

    for classRenameArg in classRen.Arguments:
      if not (classRenameArg isa BinaryExpression):
        raise "invalid class renaming expression: " + classRenameArg
```

```
expr = cast(BinaryExpression, classRenameArg)

if not (expr.Operator == BinaryOperatorType.ShiftRight \
  and expr.Left isa ReferenceExpression):
  raise "invalid class renaming expression: " + classRenameArg

fromName = cast(ReferenceExpression, expr.Left).Name

if expr.Right isa ReferenceExpression:
  toName = cast(ReferenceExpression, expr.Right).Name
  classRenaming[fromName] = toName
elif expr.Right isa MethodInvocationExpression \
  and (expr.Right as MethodInvocationExpression).Target \
  isa ReferenceExpression:

  memRen = cast(MethodInvocationExpression, expr.Right)
  toName = cast(ReferenceExpression, memRen.Target).Name
  classRenaming[fromName] = toName

  memRenaming = {}
  memberRenaming[fromName] = memRenaming

  for memRenameArg in memRen.Arguments:
    if not (memRenameArg isa BinaryExpression):
      raise "invalid member renaming expression: " + memRenameArg
    memRenExpr = cast(BinaryExpression, memRenameArg)

    if not (memRenExpr.Operator == BinaryOperatorType.ShiftRight
      and memRenExpr.Right isa ReferenceExpression):
      raise "invalid renaming expression: " + memRenameArg

    /* The left hand side of renaming can be a method name only
       or a complete method signature. In the first case, the
       method name will be represented with a simple
       ReferenceExpression, but in the latter the signature will
       be represented as a MethodInvocationExpression, and since
       such an expression can be much more compliated than a
       simple signature, we must check that the syntax of the
       MethodInvocationExpression confroms with that of a method
       signature. */
    if memRenExpr.Left isa ReferenceExpression:
      pass
    elif memRenExpr.Left isa MethodInvocationExpression:
      methodInvoc as MethodInvocationExpression = memRenExpr.Left
      for arg in methodInvoc.Arguments:
        while arg isa MemberReferenceExpression:
          arg = (arg as MemberReferenceExpression).Target
        if not (arg isa ReferenceExpression):
          raise "invalid renaming expression: " + memRenameArg
    else:
      raise "invalid renaming expression: " + memRenameArg

    memToName = cast(ReferenceExpression, memRenExpr.Right).Name
```

```
          memRenaming[memToName] = memRenExpr.Left
        else:
          raise "invalid member renaming expression: " + classRenameArg
    else:
      raise "invalid template reference: " + args[0]


/* Remove empty constructors in a class. Such might have been
   automatically during template compilation, and might cause
   unwarranted conflicts when merging template classes. */
private def RemoveEmptyConstructors(classDef as ClassDefinition):
  for member in classDef.Members:
    if (member isa Constructor):
      constr as Constructor = member

      if constr.Parameters.IsEmpty and constr.Body.IsEmpty:
        classDef.Members.Remove(constr)

      /* We consider a constructor just calling super as empty. */
      if constr.Parameters.IsEmpty and constr.Body.Statements.Count == 1 \
        and constr.Body.LastStatement isa ExpressionStatement:
        exprStmt as ExpressionStatement = constr.Body.LastStatement
        if exprStmt.Expression isa MethodInvocationExpression:
          methodInvoc as MethodInvocationExpression = exprStmt.Expression
          if methodInvoc.Target isa SuperLiteralExpression \
            and methodInvoc.Arguments.Count == 0 \
            and methodInvoc.NamedArguments.Count == 0:
            classDef.Members.Remove(constr)


/* Rename class member (fields and methods) declarations according
   to the renaming specified in memberRenaming. */
private def RenameClassMembers(classDef as ClassDefinition):
  if not (memberRenaming[classDef.Name] is null):
    memRenaming as Hash = memberRenaming[classDef.Name]

    for newName in memRenaming.Keys:
      oldNameExpr = memRenaming[newName]
      oldNameMatch = false

      for memb in classDef.Members:
        if memb isa Field:
          field as Field = memb
          if oldNameExpr isa ReferenceExpression \
            and (oldNameExpr as ReferenceExpression).Name == field.Name:
            oldNameMatch = true
            field.Name = newName
        elif memb isa Method:
          method as Method = memb
          if oldNameExpr isa ReferenceExpression \
            and (oldNameExpr as ReferenceExpression).Name == method.Name:
            oldNameMatch = true
            method.Name = newName
          elif oldNameExpr isa MethodInvocationExpression:
```

```
            methodInvoc = oldNameExpr as MethodInvocationExpression
            methodInvocName = (methodInvoc.Target as ReferenceExpression).
                Name
            if methodInvocName == method.Name \
              and MatchParameters(methodInvoc.Arguments, method.Parameters)
                  :
              oldNameMatch = true
              method.Name = newName

      if not oldNameMatch:
        raise "no match for " + oldNameExpr + " >> " + newName


/* Rename class name, including references to names of this and
   other classes that should be renamed. */
private def RenameClass(classDef as ClassDefinition):
  if not classRenaming[classDef.Name] is null:
    classDef.Name = classRenaming[classDef.Name]

  for name in classRenaming.Keys:
    sourceType = SimpleTypeReference(name as string)
    targetType = SimpleTypeReference(classRenaming[name] as string)
    classDef.ReplaceNodes(sourceType, targetType)


/* Merge members from the adds clause into the instantiated class. */
private def mergeWithAddsClass(newClass as ClassDefinition, \
                               addsClass as ClassDefinition):
  if not (addsClass is null):
    /* If there's already a member with the same name in the class
       as a member from the adds clause, the adds clause member
       will override the existing member. */
    for addsMember in addsClass.Members:
      for member in newClass.Members:
        if MatchMembers(member, addsMember):
          member.Name = FindTSuperMethodName(newClass, member.Name) + "$"
          break
      newClass.Members.Add(addsMember)

    /* Resolve tsuper calls in adds members. */
    addsClass.Accept(ResolveTSuperVisitor(newClass))


/* Merge a new class, newClass, with a class, existingClass, that
   have already been processed by a preceeding inst statement. The
   two classes should have the same name, but that is not enforced
   by this method. (It should be checked before calling this
   method.) Members from the adds clause, addsClass, might override
   members in the new class. */
private def mergeWithExistingClass(newClass as ClassDefinition, \
                                   existingClass as ClassDefinition, \
                                   addsClass as ClassDefinition):
  for member in newClass.Members:
    if not (addsClass is null):
```

```
      /* If there's already a member with the same name in the class
         as a member from the adds clause, the adds clause member
         will override the existing member. */
      for addsMember in addsClass.Members:
        if MatchMembers(member, addsMember):
          member.Name = FindTSuperMethodName(newClass, member.Name) + "$"
          break

      /* FIXME: Check for name collisions between members, but not for
         methods that are overridden. */

      /* Resolve tsuper calls in adds members. */
      addsClass.Accept(ResolveTSuperVisitor(newClass))

    existingClass.Members.Add(member)


/******************** Main macro expansion method ***************/

override def Expand(macro as MacroStatement):

  /* Process the macro arguments. */
  ReadArguments(macro.Arguments)


  /* Register the module in which the inst macro is called. */
  if not (macro.ParentNode.ParentNode isa Module):
    raise "template instantiation can only be done at module level: " \
      + templateName
  parentModule as TypeDefinition = macro.ParentNode.ParentNode

  /* Create a handle to the hash table of adds clauses that have
     already been registered by the adds attribute. */
  addsClasses as Hash = {}
  if not (parentModule["adds"] is null):
    addsClasses = parentModule["adds"]

  /* Create a hash table to register all instantiated classes. */
  instClasses as Hash = {}
  if parentModule["inst"] is null:
    parentModule["inst"] = instClasses
  else:
    instClasses = parentModule["inst"]


  /* Create an instance of the class containing the string of
     template code. */
  templateCodeClassType = FindType(templateName)
  if templateCodeClassType is null:
    raise "cannot find template: " + templateName
  templateCodeClass = Activator.CreateInstance(templateCodeClassType)

  codeField = templateCodeClass.GetType().GetField("_templateCode", \
    System.Reflection.BindingFlags.NonPublic \
```

```
  | System.Reflection.BindingFlags.Instance)
templateCode = codeField.GetValue(templateCodeClass)


/* Parse template code and bind types. (Execute all steps in the
   compiler pipeline from Parsing up to and including TypeInference.) */
compiler = BooCompiler()
compiler.Parameters.Pipeline = ResolveExpressions()
compiler.Parameters.Input.Add(StringInput("code", templateCode))
/* Make sure to reference the same libraries as in the current compiler
   context. This is necessary when templates have references to external
   libraries. */
compiler.Parameters.References = CompilerContext.Current.Parameters.
    References
context = compiler.Run()
templateCodeModule as Module = context.CompileUnit.Modules[0]

/* Check that all members of the templates are class definitions.
   This have already been checked when the template was declared,
   but it won't hurt doing it again. Also remove empty constructors
   from the classes. */
for member in templateCodeModule.Members:
  if not member isa ClassDefinition:
    raise "template can only contain class definitions: " + templateName
  RemoveEmptyConstructors(member)


/* The following loops are dependent on each other and must be
   exectuted in the particular order. They cannot be combined. */

/* Rename class member (field and method) declarations. */
for member in templateCodeModule.Members:
    RenameClassMembers(member)

/* Rename classes and class references. */
for member in templateCodeModule.Members:
    RenameClass(member)

/* Rename class member (field and method) references. */
templateCodeModule.Accept(RenameReferenceVisitor())

/* Remove bindings from all template classes. This is necessary or
   the results of the adds clause processing might be invalid.
   (Specifically in the event of overrides in an adds clause and
   the tsuper facility is used.) */
templateCodeModule.Accept(RemoveBindingsVisitor())

/* Append additional members (from the adds claues), and finally
   insert all resulting classes into the AST (using yield). */
for member in templateCodeModule.Members:
  newClass as ClassDefinition = member
  existingClass as ClassDefinition = instClasses[newClass.Name]
  addsClass as ClassDefinition = addsClasses[newClass.Name]
```

```
    /* Check if a class with the same name has already been instantiated.
       If it hasn't, incorporate members from the adds clause if any,
       otherwise merge with the already instantiated class. */
    if existingClass is null:
      mergeWithAddsClass(newClass, addsClass)

      /* Register the class as instantiated. */
      instClasses[newClass.Name] = newClass

      /* The following is a workaround instead of using yield, since
         using yield to generate multiple classes apparently results
         in a module class also being created. (Not directly, but it
         will happen in the compiler step IntroduceModuleClasses
         after macros are expanded.) Fixed with git commit b4d8e9e
         (2011-10-15) to be released as Boo 0.9.5. */
      parentModule.Members.Add(newClass)
      # yield newClass

    else: // Merge with already instantiated class of the same name
      mergeWithExistingClass(newClass, existingClass, addsClass)


/* Rename constructor and member references according to their type
   bindings. Since the node entities apparently are automatically
   updated as a result of the renaming the declarations, all we need
   to do is set the node name equal to the entity name. This is really
   only useful for purposes such as pretty printing the code since the
   names are already bound correctly regardless of whether the name of
   the corresponding declaration have been changed. */
class RenameReferenceVisitor(DepthFirstTransformer):

  private def RenameNode(node as ReferenceExpression):
    if node.Entity.Name == "constructor":
      /* Strip the last part off the full name, that is the member name,
         leaving only the class name. */
      lastPoint = node.Entity.FullName.LastIndexOf(".")
      className = node.Entity.FullName.Substring(0, lastPoint)
      node.Name = className
    else:
      node.Name = node.Entity.Name

  override def OnMemberReferenceExpression(node as MemberReferenceExpression)
      :
    RenameNode(node)
    super.OnMemberReferenceExpression(node)

  override def OnReferenceExpression(node as ReferenceExpression):
    RenameNode(node)
    super.OnReferenceExpression(node)


/* Resolve tsuper calls within a class definition. This is done by
   replacing a occurences of a tsuper call with the actual name of the
   corresponding method. FIXME: Unfortunately tsuper constructors are
```

```
   not yet supported. */
class ResolveTSuperVisitor(DepthFirstTransformer):
  _classDef as ClassDefinition

  def constructor(classDef as ClassDefinition):
    _classDef = classDef

  override def OnMemberReferenceExpression(node as MemberReferenceExpression)
      :
    if node.Target isa ReferenceExpression:
      targetRefExp as ReferenceExpression = node.Target
      if targetRefExp.Name == "tsuper":
        node.Target = SelfLiteralExpression()
        node.Name = InstMacro.FindTSuperMethodName(_classDef, node.Name)
    super.OnMemberReferenceExpression(node)


/* Remove bindings, including type and name bindings, from the AST.
   This visitor will typically be invoked for the entire AST of a
   program. */
class RemoveBindingsVisitor(DepthFirstTransformer):

  private def RemoveBindings(node as Node):
    node.Entity = null
    if node isa Expression:
      (node as Expression).ExpressionType = null

  override def OnNode(node as Node):
    RemoveBindings(node)
    super.OnNode(node)

  override def OnMethod(node as Method):
    RemoveBindings(node)
    node.Locals = null
    super.OnMethod(node)
```

A.3 AddsAttribute.boo

```
namespace BooPT

import System
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast


/* Register additional fields and methods for an instantiated template
   class. */
class AddsAttribute(AbstractAstAttribute):
  /* This code relies on attributes being processed before macros. */

  /* FIXME: Adds on classes not instatiated from a template are
     silently ignored. With the current implementation this can
     probably only be chekced with a special compiler step. */

  def Apply(node as Node):
    /* FIXME: Adds should not be allowed on inner class definitions. */
    if not (node isa ClassDefinition):
      raise "adds requires class definition"

    addsClass as ClassDefinition = node
    parentModule as Module = node.ParentNode

    /* Annotate the parent module with a hash of adds classes. */
    if parentModule["adds"] is null:
      parentModule["adds"] = {}
    addsClasses as Hash = parentModule["adds"]

    /* Presently we don't allow multiple adds to the same class. */
    if not (addsClasses[addsClass.Name] is null):
      raise "multiple adds to the same class not allowed:" + addsClass.Name

    /* Register the adds class in the hash table. */
    addsClasses[addsClass.Name] = addsClass

    /* Remove the class from the parent module since it will be
       incorporated into the existing class. */
    parentModule.Members.Remove(addsClass)
```

A.4 `CompileWithTranslateGeneratedNamesStep.boo`

```
namespace BooPT

import System.Collections.Specialized
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast
import Boo.Lang.Compiler.Steps
import Boo.Lang.Compiler.Pipelines


/* Translate names containing dollar signs into valid names. Names
   containing dollar signs are not legal in Boo, but they might be
   introduced during transformation of the AST. (During macro
   expansion for example.) Thus, code that are generated (pretty
   printed) from the AST might contain names with dollar signs, and
   therefore fail to compile. So, in order to compile such code the
   dollar signs must be replaced. This compiler step wil replace any
   dollar signs in names with underscores, which in Boo are legal to
   use in names. Furthermore, potential collisions with existing names
   which also might use underscores will be avoided by appending a
   suffix to any translated names that might possibly conflict an
   existing name. The algorithm for this is crude, and the scope of
   names is not taken into account, so a translated name might get a
   suffix even when it's not really necessary.
   Moreover the algorithm depends on the fact that a name with a
   dollar sign is unique throughout the entire program. Thus,
   except for a potential conflict with an existing name, the
   translated name with or without a suffix will also be unique
   for the entire program. So we don't need to check that
   tranlstated names conflict with each other. */
class TranslateGeneratedNamesStep(AbstractTransformerCompilerStep):

  override def Run():
    ast = CompilerContext.Current.CompileUnit

    namesTableBuilder = NamesTableBuilderVisitor()
    ast.Accept(namesTableBuilder)
    namesTable = namesTableBuilder.GetTable()

    namesTranslator = TranslateGeneratedNamesVisitor(namesTable)
    ast.Accept(namesTranslator)


/* This pipeline executes all the standard compiler steps, except
    code generation, and then as the last step translates dollar
    signs to underscores in names. */
class CompileWithTranslateGeneratedNamesStep(Compile):
  def constructor():
    super()
    Add(TranslateGeneratedNamesStep())
```

```
/* Build a table of all names in the program containing underscores
   but not dollar signs. This is used in checking for conflicts with
   existing names in the program when translating dollar signs in
   names to underscores. Since a translated name always will contain
   underscores the name table doesn't need to store any names that
   doesn't contain underscores, since those cannot conflict with
   translated names. Names containing underscores, but also dollar
   signs, won't be stored either since these are names that will be
   translated later. */
class NamesTableBuilderVisitor(DepthFirstTransformer):

  _namesTable = StringCollection()

  public def GetTable():
    return _namesTable

  override def OnMemberReferenceExpression(node as MemberReferenceExpression)
      :
    if node.Name.Contains('_') and not node.Name.Contains('$'):
      _namesTable.Add(node.Name)
    super.OnMemberReferenceExpression(node)

  override def OnReferenceExpression(node as ReferenceExpression):
    if node.Name.Contains('_') and not node.Name.Contains('$'):
      _namesTable.Add(node.Name)
    super.OnReferenceExpression(node)


/* Translate names containing dollar signs by replacing the dollar signs
   with underscores in a way as to not conflict with names in a given table.

   If a translated name conflicts with (that is is equal to) a name in
   the table, an underscore followed by an integer will be appended to
   the translated name, and the integer will be increased until there
   is no conflict with names in the table. */
class TranslateGeneratedNamesVisitor(DepthFirstTransformer):

  _namesTable as StringCollection

  def constructor(namesTable as StringCollection):
    _namesTable = namesTable

  def translateName(node as ReferenceExpression):
    if node.Name.Contains('$'):
      translatedName = node.Name.Replace('$', '_')
      if not _namesTable.Contains(translatedName):
        node.Name = translatedName
      else:
        i = 1
        while _namesTable.Contains(translatedName + "_" + i):
          i++
        node.Name = translatedName + "_" + i
```

```
override def OnMemberReferenceExpression(node as MemberReferenceExpression)
    :
  translateName(node)
  super.OnMemberReferenceExpression(node)

override def OnReferenceExpression(node as ReferenceExpression):
  translateName(node)
  super.OnReferenceExpression(node)
```