Et system for å resonnere forover rundt kode skrevet i objektorienterte språk

Johan Dovland
Institutt for informatikk, Universitetet i Oslo

1 Innledning

Objektorientert programmering har utviklet seg til å bli den dominerende programmeringsmetoden. Ikke minst med programmeringsspråkene C++ og Java har objektorientering opparbeidet seg stor popularitet og nye anvendelsesområder.

Uavhengig av utviklingen rundt objektorientering, er det arbeidet med å utarbeide metoder og bevissystemer for å resonnere rundt, og verifisere programmer. Slike systemene gir mulighet til å formelt avgjøre hvorvidt implementasjoner tilfredstiller gitte spesifikasjoner. Dette har praktisk nytte i for eksempel programdokumentasjon og sikkerhetsanalyse.

Et mye brukt verifikasjonssystem i så måte ble introdusert av C.A.R. Hoare i 1969, og kalles *Hoare-logikk* [8]. Tradisjonell Hoare-logikk er først og fremst velegnet til å vise korrekthet av programkode i imperative språk uten objekter og objektpekere, for eksempel Algol. Når man forsøker å bruke tradisjonell Hoare-logikk i en objektorientert sammenheng, oppstår det problemer. Den type problemer vi skal se på her, oppstår som følge av *pekeraliasing*. Pekeraliasing oppstår når to pekeruttrykk referer til samme objekt. Vi skal her se på hva som kan gjøres for å utvide Hoare-logikk til systemer der pekere er tillatt.

I tillegg til å håndtere pekeraliasing, er vi interresert i best mulig å bevare flere av de sentrale egenskapene til Hoare-logikk, slik som enkelhet og automatiserbarhet. Hoare-logikk er et enkelt aksiomatisk bevissystem, og denne enkelheten er vi interresert i å bevare best mulig. Automatisering øker anvendbarheten til systemet, fordi det blir mulig å generere bevis uten manuell bearbeiding. Selv om man utvikler aksiomer og systemer som er korrekte og behandler pekere riktig, minker verdien til systemet hvis det i praksis blir vanskelig å forstå, eller komplisert å bruke.

En stor del av de programmene som skrives i dag, skrives i objektorienterte språk. Dette fører med seg pekeraliasing, så for å kunne verifisere mange av programmene som skrives i dag, trenger vi systemer for å verifisere kode der aliasing er mulig.

Flere har interresert seg for problemstillinger rundt Hoare-logikk og pekeraliasing. Luckham og Suzuki [10], utarbeider et system for verifikasjon av programmer i Pascal. Når man resonnerer rundt tilordninger, er det med utgangspunkt i de tilordningsaksiomene som finnes i Hoare-logikk, mest praktisk å arbeide bakover. Morris [11] og Bornat [3] viser hvordan bakoverkonstruksjon kan utvides til å håndtere tilordninger til pekeruttrykk. I denne artikkelen vil vi derimot se på hvordan vi kan arbeide forover over tilordninger. Dette gjøres ved å

beholde den gjeldende Hoare-logikken i størst mulig grad, og legge på sidebetingelser i reglene som vedlikeholder sunnhet når vi resonnerer over tilordninger. Strategien vil også bli sammenliknet med bakoverresonnering.

Når vi resonnerer forover, mister vi muligheten for å finne svakeste forbetingelse til kode. Dette oppveies imidlertid av muligheten vi får til å finne sterkeste bakbetingelse. Bakbetingelsen vi får ved forover bevisføring inneholder all aliasinformasjonen vi kan ha på bakgrunn av forbetingelsen til tilordningene.

2 Bakgrunn

Hoare introduserte tripler (gjerne kalt Hoare-tripler), på formen {P}S{Q}, der S er en eller flere setninger i vårt programmeringspråk, og P og Q er uttrykk i et formålstjenlig predikatspråk. P kalles forbetingelsen til S, og Q kalles bakbetingelsen. Idéen er at P inneholder informasjon om tilstanden til programmet før utførelsen av S, og Q inneholder informasjon om tilstanden etter utførelsen av S.

Dersom vi for eksempel har gitt en heltallsvariabel x med verdien 5, kan dette skrives som predikatet $\{x = 5\}$. Dersom vi har en programsetning x:=x+1, blir betingelsen etter utføring $\{x = 6\}$. Dette kan vi formulere med Hoare-trippelet $\{x = 5\}$x:=x+1$\{x = 6\}$.

Programmeringsspråket

Vi bruker et enkelt programmeringsspråk med dotnotasjon. Dessuten har språket tilordninger og if-setninger. Verdilikhet av uttrykk testes v.h.a. operatoren '='. To pekeruttrykk er verdilike dersom de referer til (peker på) samme objekt. Vi konsentrer oss om de imperative delene av språket, ikke de deklarative. Alle uttrykk antas derfor å være typekorrekte og alle variable som brukes antas å være deklarerte og typete. For å holde språket enkelt, tillater vi ikke subklasser.

Predikatspråket

Vi bruker førsteordens predikatspråk med likhet og udefinerthet av uttrykk. Udefinerte uttrykk betegnes det med stor omega, Ω. Alle uttrykk i programmeringsspråket er også uttrykk i predikatspråket. I tillegg kan vi bygge opp uttrykk ved hjelp av de vanlige logiske operatorene ikke($\neg$), og($\wedge$), eller($\vee$) og implikasjon($\Rightarrow$), samt egendefinerte funksjoner.

For å takle udefinerthet av pekeruttrykk, bruker vi treverdilogikk (**t**, **f**, Ω) som hos Kleene[9].

Likhet i predikatspråket er strikt, slik at $\Omega = x$ blir Ω. Vi innfører et if-uttrykk på formen (**if** *B* **then** *A* **else** *C* **fi**). Dette kan tolkes på samme måte som uttrykket (B $\wedge$ A) $\vee$ ($\neg$B $\wedge$ C). Dette gir oss den velegnede egenskapen at (**if false then** Ω **else** *C* **fi**) og (**if true then** *A* **else** Ω **fi**) er veldefinert. Den første er ekvivalent med C og den andre med A. Vi bruker også allkvantor $\forall x$, og eksistenskvantor $\exists x$. Likhet binder sterkest og kvantorene svakest. Vi innfører også $\Longleftrightarrow$ som er akkurat det samme som =, bortsett fra at den binder svakere enn kvantorene.

Gitt en variabel x, og et uttrykk P i predikatspråket. Vi sier at x er *bundet* i P dersom den er kvantorisert i P. At x er kvantorisert i P, betyr at alle forekomster av x ligger innenfor en kvantorisert del av uttrykket. Gitt for eksempel at $\forall x \cdot$P′

eller $\exists x \cdot \mathrm{P}'$ forekommer i P. Dersom x ikke finnes noen andre steder i P enn inne i P', er x bundet i P.

Variable som ikke er bundet i P, er *frie*. Mengden av frie variable i P betegnes med $\mathcal{V}[\mathrm{P}]$. En formel(P) uten noen frie variable ($\mathcal{V}[\mathrm{P}] = \emptyset$) kalles en lukket formel.

Gitt et predikatuttrykk P, programvariabel x og uttrykk i programspråket e. Med uttrykket P^x_e menes det samme som P, men der alle frie forekomster av x er syntaktisk byttet ut med verdien av uttrykket e.

Predikatlogikken

Når vi gjør utledninger i Hoare-logikk, får vi verifikasjonsbetingelser. Disse betingelsene er uttrykk i predikatspråket vårt, og kan ikke bevises i Hoare-logikk. For å utlede disse, trenger vi en kalkyle for predikatspråket. Eksempler på slike er gitt av Dahl[6]. Det kan for eksempel være naturlig deduksjon (ND) eller BPC justert for udefinerthet (WS-logikk)[12]. Disse systemene er bygd opp av aksiomer og regler. Setningene i predikatlogikken er på formen $A \vdash B$. Disse er bygd opp av sekvent-tegn($\vdash$) og to uttrykk. Ved å anta det som står foran sekventtegnet (A), skal vi vise det som kommer etter (B). Uttrykket A kalles antesedenten, og B kalles suksedenten.

Det grunnleggende aksiomet er:

$\mathrm{P} \vdash \mathrm{P}$, *der* P *er et vilkårlig veldefinert uttrykk.*

Det er lett å overbevise seg om at dette er riktig. Ved å anta P, skal vi vise P.

Den første ordens predikatkalkylen må også håndtere likhet. Pekerlikhet er viktig, så vi ser nærmere på noen av disse reglene. For en variabel x av typen T, ser aksiomet for likhet ser slik ut:

EAX: $\forall x{:}\mathrm{T} \cdot x = x$, *der domenet til typen T er veldefinert*

Vi har også reglene $\forall$E og TEQ:

$$\forall E : \frac{\forall x : T \ \cdot \ P(x)}{P(t)} \ , \quad t : T$$

$$\mathrm{TEQ} : \frac{\vdash P^{\alpha}_{t'}; \quad \vdash t = t'}{\vdash P^{\alpha}_{t}}$$

Gitt at vi kan bevise $t = t'$, og har ett uttrykk P, som skal verifiseres, så kan vi substituere bort et vilkårlig antall t med t'. Det trenger ikke være alle, og α er ment til å markere hvilke.

Definerthet av pekeruttrykk

En peker referer vanligvis til (peker på) et objekt. Når vi ønsker å si at en peker ikke peker på et objekt, bruker vi verdien *nil*. Dette tilsvarer Javas `NULL`.

Alle pekeruttrykk som ikke omtaler *nil* antas å være definerte, og evaluerer til eksisterende objekter. En peker som har verdien *nil* er veldefinert. Å bruke dotnotasjon til å dereferere en *nil*-peker er derimot ikke veldefinert.

For eksempel hvis m = *nil* så er m veldefinert, mens $m.E$ ($nil.E$) er derimot udefinert for vilkårlig uttrykk E.

Med utgangspunkt i dette defineres definerthetsoperatoren $\mathbf{d}_E$ ved strukturell induksjon over E slik:

$\mathbf{d}_x \equiv \mathbf{true}$

$\mathbf{d}_{x.E} \equiv \mathbf{if}\ x = nil\ \mathbf{then\ false\ else}\ \mathbf{d}_E\ \mathbf{fi}$

Uttrykket x er programviariabel.

Hoare-logikk

Hoare-logikk gir et aksiomatisk system for å vise korrekthet av programkode. Det vil si at vi på tilsvarende måte som i predikatkalkylene ND, BPC og sekventkalkyle fra [6] har ett eller flere aksiomskjemaer og at det finnes sett med utledningsregler som brukes til bevisføring.

Aksiomskjemaer

Først ser vi på aksiomskjemaer som gjelder for tilordninger. Hoare[8] og Dahl[6] presenterer samme aksiomskjema for tilordning til programvariabel(x).

AS: $\{\mathrm{P}^{\mathrm{x}}_{\mathrm{e}}\}\mathtt{x} := \mathtt{e}\{\mathrm{P}\}$, *for veldefinert P og e*

I AS er bakbetingelsen(P) et vilkårlig uttrykk, og forbetingelsen fåes ved å substituere verdien til uttrykket e for x i P. Forbetingelsen konstrueres altså fra bakbetingelsen.

Tilsvarende gir Dahl et aksiomskjema til bruk ved forover bevisføring. Her starter vi med en vilkårlig forbetingelse(P), og fører denne fremover over tilordningene. Dette gjør at bakbetingelsen formuleres på grunnlag av forbetingelsen.

RCAS: $\{P\}\mathtt{x}:=\mathtt{e}\{\exists\alpha \cdot P^x_\alpha \wedge \mathbf{d}_{e^x_\alpha} \wedge x = e^x_\alpha\}$, *der* $\alpha \notin \mathcal{V}[P, e, x]$ *og P veldefinert*

Uttrykket α kan vi tenke på som verdien til x før tilordningen. Bakbetingelsen sier da at P forsatt gjelder med den gamle verdien til x. Dessuten har x fått sin nye verdi(e) basert på verdien x hadde før tilordningen, slik vi er vant til fra språk som Java og C.

Utledningsregler

Utledningsreglene i Hoare-logikk (Hoare-regler) er på formen:

$$\frac{\mathrm{R}_1; \mathrm{R}_2; \ldots; \mathrm{R}_\mathrm{n}}{\mathrm{R}}$$

Det finnes forskjellige typer regler. I en Top-Down (TD) regel starter vi med trippelet vi ønsker å bevise, og konstruerer beviset ved å finne separate delbevis for hver av $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_\mathrm{n}$. Når en regel brukes i en Bottom-Up (BU) strategi, har vi separate bevis for $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_\mathrm{n}$ og utleder konklusjonen fra disse.

I tillegg til BU og TD har vi to andre bevisstrategier. Disse kalles Left-Construction (LC) og Right-Construction (RC). Gitt en bakbetingelse til programsetninger, bruker vi LC-strategi til å arbeide seg frem mot en forbetingelse. Et mye brukt eksempel på dette er bruk av AS for å resonnere bakover over flere

påfølgende tilordninger. Med RC-strategi går vi motsatt vei, vi starter med en forbetingelse, og konstruerer bakbetingelsen. Et eksempel på dette er bruk av RCAS for å resonnere forover over tilordninger.

For å få helhet i utledninger, trenger vi reglene CQL, CQR og SEQ.

$$\text{CQL}: \frac{\vdash P \Rightarrow P';\ \vdash \{P'\}S\{Q\}}{\vdash \{P\}S\{Q\}}$$

$$\text{CQR}: \frac{\vdash \{P\}S\{Q'\};\ \vdash Q' \Rightarrow Q}{\vdash \{P\}S\{Q\}}$$

$$\text{SEQ}: \frac{\vdash \{P\}S_1\{Q\};\ \vdash \{Q\}S_2\{R\}}{\vdash \{P\}S_1;S_2\{R\}}$$

Formelene $P \Rightarrow P'$ og $Q' \Rightarrow Q$ er verifikasjonsbetingelser som er formler i predikatspråket. Disse bevises ved hjelp av predikatkalkyle.

Disse reglene kan brukes begge veier, både ovenfra og ned, eller nedenfra og opp.

Intuisjonen er kort fortalt at hvis vi ønsker å bevise $\{P\}S\{Q\}$, kan CQL brukes til å svekke forbetingelsen fra P til P′, og heller bevise $\{P'\}S\{Q\}$. Da er tilleggskravet fra CQL at vi kan bevise P′ gitt P. Tilsvarende for CQR, her styrker vi bakbetingelsen fra Q til Q′ når vi bruker regelen oppover. SEQ kan brukes ovenfra og ned til å sette sammen Hoare-tripler til større bevis, eller nedenfra og opp til å splitte et større bevis opp i mindre delbevis.

Vi formulerer en RC-regel for if-setninger. Ved RC-regel skal vi starte med forbetingelsen, og konstruere bakbetingelsen. Regelen er ikke gitt av Dahl, og vi kaller den RCIF:

$$\text{RCIF}: \frac{\vdash \{P \wedge B\}S_1\{Q_1\};\ \vdash \{P \wedge \neg B\}S_2\{Q_2\}}{\vdash \{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\{Q_1 \vee Q_2\}}$$

Denne regelen brukes ovenfra og ned.

Dette utgjør de mest sentrale delene av bevissystemet vårt, og kan greit utvides med andre regler, for eksempel for løkker.

Det programmeringsspråket vi har sett, omfatter en begrenset del av de programkonstruksjonene det finnes Hoare-regler for. Det finnes for eksempel Hoare-logikk som omfatter Input/Output av verdier i programmet, parallellitet og så videre. Problemene som oppstår ved å innføre pekere i språket, er så grunnleggende, Hoares tilordningsaksiom holder ikke, at et stort språk ikke er nødvendig for å arbeide med problemene. Det første steget på veien mot et system som behandler aliasing korrekt, er å få godt system for det grunnleggende språket.

Et Hoare-trippel er gyldig dersom det er veldefinert og sant for tilfeldige typekorrekte verdier av sine frie variable, om noen, og for vilkårlige tolkninger av funksjonssymbolene. Tolkningene av funksjonssymbolene må stemme overens med profilene deres, og eventuelle aksiomer om symbolene. Tolkningen av Hoare-trippelet $\{P\}S\{Q\}$ er at dersom P holder før eksekveringen av S, og eksekveringen av S terminerer normalt, så vil Q holde etter eksekveringen.

Tradisjonell Hoare-logikk har to viktige egenskaper, nemlig at den er sunn og komplett relativt til programmeringsspråkets semantikk [1, 5]. At Hoare-logikk

er sunn betyr at dersom vi har bevist et Hoare-trippel, så vet vi at dette trippelet også er gyldig. Kompletthet betyr at det finnes bevis for alle gyldige tripler.

Ved å innføre objektorientering i språket, mister Hoare-logikken sunnhet (og kompletthet). Dette kan vi begrunne ut fra problemer som følge av pekeraliasing, og vil bli nærmere beskrevet under.

Aliasing

Generelt sier vi at aliasing oppstår når to forskjellige uttrykk i programmeringsspråket referer til samme minnelokasjon. Aliasing kan ha flere former, men her konsentrer vi oss om pekeraliasing.

Pekeraliasing oppstår i programmer skrevet i objektorienterte språk når to pekere peker på samme objektet, eller mer presist når to forskjellige pekeruttrykk evaluerer til samme verdi, d.v.s. at de refererer til samme minnelokasjon.

Eksempel 1. **Eksempler på pekeraliasing.**
Gitt to pekervariable, x og y. Dersom x og y peker på samme objekt, er x og y aliaser for hverandre. Dersom objektet de peker på har en komponent v, vil også $x.v$ og $y.v$ være aliaser. Dersom v peker tilbake på objektet det er komponent i, seg selv, er det uendelig mange aliaser i systemet. For eksempel er x, $x.v$, $y.v$ og $x.v.v.v.v$ noen av disse.

Slutt eksempel.

Denne formen for aliasing vil ikke oppstå med enkle variable (f.eks. heltall eller boolske), fordi to syntaktisk forskjellige variable alltid vil referere til forskjellige minnelokasjoner. Dermed kan vi bruke AS og vanlig Hoare-logikk til å resonnere rundt programkode. Vanlig bruk av AS er ikke alltid vellykket i språk som tillater pekere til objekter. Dette ser vi nærmere på under.

Et problem ved aliasing, er at eksekveringen av en metode kan endre på tilstanden til objekter som tilsynelatende ikke er involvert. Gitt for eksempel to pekere x og y som peker på samme objekt, og at dette objektet har en komponent v. Tilordningen `x.v:=y` vil endre på $y.v$ uten at $y.v$ er nevnt i koden. Uventede effekter og problemer kan oppstå dersom programmerer ikke er klar over eller ikke tenker over at x og y kan peke på samme objekt.

Endringen av tilstanden til et objekt kan til og med skje uten at det affiserte objektet aksesseres. Det er fordi tilstanden til et objekt ikke bare er beskrevet av variablene i objektet. Tilstanden er også avhengig av tilstanden til alle objektene som kan nåes via variablene. Tilstanden til et objekt er derfor beskrevet av tilstanden til den transitive tillukningen av alle objekter som kan nåes fra det. Dersom vi har to forskjellige objekter, og det finnes minst ett objekt som kan nåes fra begge disse, så oppstår aliasing. Aliasing er altså ikke syntaktisk avgjørbart.

Det finnes flere former for tilordning som omhandler pekere. Det første er å sette en pekervariabel (x) til å peke på objektet som finnes ved å evaluere uttrykket E. Dette skrives `x:= E`. Uttrykket E kan inneholde dotnotasjon. Programsetningen `x:=y.a` uttrykker at x tilordnes verdien av a-komponenten i objektet y referer til.

Den andre formen for tilordning som omfatter pekere er tilordning til en komponent(variabel) i et objekt, `x.y:= z`. Her går en inn i objektet som x peker på, og gjør oppdatering på y-komponenten (variabelen y i objektet).

Vanlig Hoare-logikk fungerer når vi tilordner til programvariabel (x). Dette gjelder selv om x er en objektpeker! Før tilordningen kan det finnes flere aliaser til x, men det betyr bare at det finnes flere pekere som peker på objektet x peker på. Ingen av disse pekerne kan endres som følge av oppdatering til x.

Når vi tilordner til objektkomponenter kan vi derimot ikke regne med at vanlig Hoare-logikk fungerer. I de enkleste eksemplene går det bra, men det skal ikke mye til før vi kommer i vanskeligheter. Grunnen til dette har vi beskrevet, $x.n$ vil endres som en følge av tilordning til $y.n$ dersom x og y peker på samme objekt.

Rent formelt kan de problemene Hoare-logikk får, illustreres med et lite eksempel.

Eksempel 2.
I tradisjonell Hoare-logikk er trippelet $\{x.a = 5\}\texttt{y.a:=y.a+1}\{x.a = 5\}$ bevisbart. Ved bruk av AS kan bakbetingelsen tilbakeføres over tilordningen, og vi får aksiomet $\{(x.a = 5)^{y.a}_{y.a+1}\}\texttt{y.a:=y.a+1}\{x.a = 5\}$.

I tillegg forlanger verifikasjonsbetingelsen til CQL at vi må vise $x.a = 5 \Rightarrow (x.a = 5)^{y.a}_{y.a+1}$. Denne lar seg greit bevise. $x.a$ er ikke syntaktisk lik $y.a$ slik at hele betingelsen reduseres til $x.a = 5 \Rightarrow x.a = 5$, som er triviell.

Vi kan imidlertid tenke oss at x og y peker på det samme objektet. Da skjønner vi ut fra sammenhengen at $x.a$ skal endres som følge av tilordningen, og at trippelet vi startet med er ugyldig. Imidlertid er trippelet forsatt bevisbart! Også trippelet $\{x = y \wedge x.a = 5\}\texttt{y.a:=y.a+1}\{x.a = 5\}$ er bevisbart med ren syntaktisk substitusjon.

Slutt eksempel.

3 Metode for forover resonnering

Som en følge av muligheten for «skjulte» aliaser, blir substitusjonsmekanismen viktig. Det er ikke nok å utføre vanlig syntaktisk substitusjon når vi resonnerer rundt tilordning til objektkomponenter. Med bakgrunn i denne problematikken, gir Morris [11] gir et aksiomskjema for tilordning der han resonnerer bakover. Morris foretar ikke en syntaktisk substitusjon slik vi er vant med fra vanlig Hoare-logikk, men en semantisk. Denne substitusjonsmekanismen er senere tatt opp og videreutviklet av Bornat [3].

Når vi tilordner til objektkomponenten $x.n$, må vi ta hensyn til at det kan finnes flere pekere til objektet x refererer til. Gitt en peker y til et objekt. tilordningen til $x.n$ kan ikke affisere andre en komponenten n i y. F.eks. vil en eventuell komponent $y.p$ være uendret som en følge av tilordningen. Komponenten $y.n$ vil derimot være endret dersom x og y referer til samme objekt. Dette sammenfattes i substitusjons-definisjonene:

$$(E'.p)^{f.n}_{e} \equiv ((E')^{f.n}_{e}).p$$
$$(E''.n)^{f.n}_{e} \equiv \textbf{if } (E'')^{f.n}_{e} = f \textbf{ then } e \textbf{ else } ((E'')^{f.n}_{e}).n \textbf{ fi}$$

Bornat argumenterer grundig for at denne mekanismen er riktig, og det er også denne mekanismen vi kommer frem til i en eksempelstudie utført i [7].

Eksempel 3.
La oss se på verifikasjonsbetingelsen fra eksempel 2.

Uttrykket $x = y \wedge x.a = 5 \Rightarrow (x.a = 5)^{y.a}_{y.a+1}$ vil med den semantiske substitusjonsmekanismen bli omskrevet slik:
$x = y \wedge x.a = 5 \Rightarrow (x.a = 5)^{y.a}_{y.a+1} \Longleftrightarrow$
$x = y \wedge x.a = 5 \Rightarrow \textbf{if } x^{y.a}_{y.a+1} = y \textbf{ then } y.a + 1 = 5 \textbf{ else } x^{y.a}_{y.a+1}.a = 5 \textbf{ fi} \Longleftrightarrow$
$x = y \wedge x.a = 5 \Rightarrow \textbf{if } x = y \textbf{ then } y.a + 1 = 5 \textbf{ else } x.a = 5 \textbf{ fi} \Longleftrightarrow$
$x = y \wedge x.a = 5 \Rightarrow y.a + 1 = 5$
Dette uttrykket er ikke bevisbart, slik vi ønsker.

Hvis vi substituerer e med $x.n$ over et uttrykk som har en annen komponentnavn enn n på ytterste nivå, f.eks. $y.p$ skyves substitusjonen inn på y slik:
$(y.p)^{x.n}_{e} \Longleftrightarrow y^{x.n}_{e}.p \Longleftrightarrow y.p$

Slutt eksempel.

Den semantiske substitusjonsmekanismen fører imidlertid til andre problemer. Gitt at vi vil substituere inn v for $u.s$ i uttrykket $p.s.s = v$. Dette går slik:

$$
\begin{aligned}
(p.s.s = v)^{u.s}_{v} &\Longleftrightarrow \textbf{if } (p.s)^{u.s}_{v} = u \textbf{ then } v \textbf{ else } (p.s)^{u.s}_{v}.s \textbf{ fi} = v \\
&\Longleftrightarrow ((p.s)^{u.s}_{v} = u) \vee ((p.s)^{u.s}_{v} \neq u \wedge (p.s)^{u.s}_{v}.s = v) \\
&\Longleftrightarrow (\textbf{if } p = u \textbf{ then } v \textbf{ else } p.s \textbf{ fi} = u) \\
&\qquad \vee \\
&\qquad ((\textbf{if } p = u \textbf{ then } v \textbf{ else } p.s \textbf{ fi} \neq u) \\
&\qquad\ \wedge (\textbf{if } p = u \textbf{ then } v \textbf{ else } p.s \textbf{ fi} = u).s = v) \\
&\Longleftrightarrow (p = u \wedge v = u) \vee (p \neq u \wedge p.s = u) \\
&\qquad \vee \\
&\qquad (p = u \wedge v \neq u \wedge v.s = v) \vee (p \neq u \wedge p.s \neq u \wedge p.s.s = v)
\end{aligned}
$$

Her ser vi tydelige forskjeller på syntaktisk og semantisk substitusjon. Ved syntaktisk substitusjon får vi ikke den eksplosjonen i størrelsen på predikatene vi ser her. Dette problemet gjør seg for alvor gjeldende når vi substituerer bakover over flere tilordninger. Da må vi gjerne arbeide helt tilbake til forbetingelsen før vi kan utnytte aliasinformasjonen denne gir til å forkorte predikatet.

Systemet vårt tar utgangspunkt i den semantiske substitusjonsmekanismen som er gitt over. For å resonnere over tilordninger bruker vi RCAS, men studien utført i [7] viser at vi ofte er i spesialtilfeller som gjør at RCAS kan forenkles. De skjemaene vi gir setter opp enkle betingelser som må være oppfylt, og disse betingelsene sikrer sunnhet i resonneringen.

Aksiomskjemaer for forover resonnering

Aksiomskjemaet RCAS er bunnen i metoden for forover resonnering. Gitt en enkel programvariabel, x, og et uttrykk e som godt kan inneholde dotnotasjon. Da har vi RCAS som før:

RCAS: $\{P\}\mathtt{x:=e}\{\exists\alpha \cdot P^x_\alpha \wedge \mathbf{d}_{e^x_\alpha} \wedge x = e^x_\alpha\}$, *der* $\alpha \notin \mathcal{V}[P, e, x]$ *og* P *veldefinert*

Gitt x enkel programvariabel, så kan vi gi to forenklinger av RCAS (Short-RCAS) slik:

SRCAS : $\vdash \{\mathrm{P} \wedge\ x = a\}\mathtt{x:=e}\{\mathrm{P}^x_a \wedge\ x = e^x_a\}$, x $\notin \mathcal{V}$[a] og e veldef.
SRCAS′ : $\vdash \{\mathrm{P}\}\mathtt{x:=e}\{\mathrm{P} \wedge\ x = e\}$, x $\notin \mathcal{V}$[P, e] og e veldef.

Når det gjelder tilordning til objektkomponenter, foreslår vi følgende forenklinger [7]. Her er x en enkel programvariabel, det vil si uten dotnotasjon. Skjemaet kaller vi Component-RCAS, og også dette har en merket versjon:

CRCAS: $\vdash \{P \wedge x.n = a\}\texttt{x.n:=e}\{P_a^{x.n} \wedge x.n = e_a^{x.n}\}$
der a ikke omtaler n-komponenter, og e veldef.

CRCAS′: $\vdash \{P\}\texttt{x.n:=e}\{P \wedge x.n = e\}$
der P og e ikke omtaler n-komponenter, og e veldef.

At et uttrykk P ikke omtaler n-komponenter, er det samme som å sjekke hvorvidt uttrykket $E.n$ forekommer i P for vilkårlig E. Dette er en ren syntaktisk sjekk.

Sidebetingelsen til CRCAS kan slakkes på, men det vil gå på bekostning av enkelheten. Vi kan tillate at a omtaler n-komponenter. Dersom $y.n$ forekommer i a, må vi da være sikret $x \neq y$ fra P. Dette er ikke syntaktisk avgjørbart.

For mange tilordningene kan vi bruke CRCAS med sidebetingelsen slik det står. Dersom vi ikke kan bruke noen av de forenklede reglene, må vi bruke RCAS direkte. For eksempel dersom vi ikke klarer å oppfylle sidebetingelsene, eller at vi ikke har noen aliaser for uttrykket vi tilordner til.

Her har vi presentert tilordninger til uttrykk med enkel dotnotasjon. I og med at vi kan bruke hjelpevariable til å omskrive uttrykk med flere dottinger til uttrykk med enkel dot, er dette stammen i systemet vårt. Vi kan imidlertid bruke CRCAS også ved tilordninger til uttrykk med flere dottinger. For tilordninger på formen `x.n.p :=e` fungerer CRCAS greit fordi objektkomponentnavnene n og p er tekstlig ulike. (x er altså forsatt en enkel programvariabel.) Dette kan vi formulere som skjemaet Double-Component-RCAS:

DCRCAS: $\vdash \{P \wedge x.n.p = a\}\texttt{x.n.p:=e}\{P_a^{x.n.p} \wedge x.n.p = e_a^{x.n.p}\}$
der a ikke omtaler p-komponenter, og e veldef.

På tilsvarende måte som for CRCAS, kan sidebetingelsen slakkes. Vi kan tillate at a inneholder deluttrykk på formen $y.p$, men da må vi være sikret at y og $x.n$ ikke peker på samme objekt.

Tilordningen `x.n.n := e` er den andre formen for tilordning til uttrykk med dobbel dotnotasjon. Her forekommer komponentnavnet vi gjør tilordning til flere ganger i uttrykket. I følge skjemaene vi har sett så langt, skal vi da kunne sette opp $\{x.n.n = e_\alpha^{x.n.n}\}$ som siste konjunkt i bakbetingelsen. Dette er ikke sunt dersom $x = x.n$ før tilordningen! I [7] foreslår vi å dele dette i to tilfeller. Dersom vi har $x \neq x.n$ i forbetingelsen kan vi bruke DCRCAS. Dersom vi har $x = x.n$ i forbetingelsen, kan vi bruke følgende skjema (x er enkel programvariabel):

DCRCAS′: $\vdash \{P \wedge x = x.n\}\texttt{x.n.n:=e}\{P_x^{x.n} \wedge x.n = e_x^{x.n}\}$

For å konstruere bevis av imperative programmer, vil vi i tillegg til aksiomskjemaene trenge Hoare-reglene vi gav tidligere, samt predikatlogikk.

Sunnhet av aksiomskjemaene

Vi kan argumentere for sunnhet av de forenklede aksiomskjemaene våre. Argumentasjonen går likt for hver regel, så vi går bare i detalj for en regel, CRCAS.

CRCAS gjelder for setninger på formen `x.n:=e`, tilordning til objektkomponent. Før tilordningen har vi $\{\mathrm{P} \wedge x.n = a\}$, og kravet til regelen er at a ikke skal omtale n-komponenter.

Siden $x.n = a$, er P=$\mathrm{P}_a^{x.n}$ før tilordningen. Her er $\mathrm{P}_a^{x.n}$ disjunkt fra $x.n$, slik at $\mathrm{P}_a^{x.n}$ ikke endres av tilordningen. Denne kan dermed skyves gjennom tilordningen og inn i bakbetingelsen.

Fra likheten $x.n = a$, har vi også at $e = e_a^{x.n}$. Her er $e_a^{x.n}$ uavhengig av $x.n$, slik at verdien av dette ikke endres av tilordningen. Det er denne verdien som tilordnes til $x.n$, slik at $x.n$ har verdien $e_a^{x.n}$ etter tilordningen. Hele bakbetingelsen blir dermed $\{\mathrm{P}_a^{x.n} \wedge x.n = e_a^{x.n}\}$

Motivasjon for å resonnere forover

Ofte foretrekker man å gjøre Hoare-analyse bakover over tilordninger. Tradisjonelt kan dette forsvares ut fra enkelheten av aksiomskjemaet AS i forhold til RCAS.

Noe av det viktigste for Hoare-analyse rundt objektorienterte språk er at den vanlige syntaktiske substitusjonsmekanismen ikke lenger holder når en gjør tilordning til objektkomponenter. Bornat foreslår et system for bakoveranalyse, og bruker den samme semantiske substitusjonsmekanismen som vi har sett her. Med utgangspunkt i dette og det systemet vi har lagt frem her, kan vi sette opp flere punkter som argumenterer for at foroveranalyse er vel så attraktivt som bakoveranalyse.

Enkelhet i bruk av skjemaene. I utgangspunktet er AS det enkleste aksiomskjemaet i bruk. Dette kommer av RCAS sin innføring av eksistenskvantor i bakbetingelsen. Når vi bruker RCAS, er vi imidlertid ofte i spesialtilfeller som gjør at vi ikke trenger å bruke denne kvantoren. Da blir bruk av RCAS like enkel som bruk av AS. Aliasinformasjonen i forbetingelsen er avgjørende for bruk av disse reglene.

Uttrykkskraften i predikatene. Når vi resonnerer forover, tar vi all aliasinformasjonen fra forbetingelsen med til bakbetingelsen. I tillegg legger til den nye informasjonen vi får fra tilordningen. Når vi derimot resonnerer bakover over tilordning til variable, vil ikke forbetingelsen inneholde aliasinformasjon om variabelen vi gjør tilordning til. Dette skjer fordi vi substituerer bort alle forekomster av uttrykket vi tilordner til. For eksempel i skjemaet AS som er satt opp tidligere, inneholder ikke forbetingelsen $\mathrm{P}_{\mathrm{e}}^{\mathrm{x}}$ noen informasjon om x.

Forenkling av predikatene underveis i beviset. Ved forover resonnering kan vi gjøre bruk av aliasinformasjonen vi har fra forbetingelsen underveis i beviset. Dette gjør at predikatene kan forenkles underveis i beviset. Aliasinformasjonen fra forbetingelsen er nemlig tilgjengelig gjennom hele beviskonstruksjonen. Aliasinformasjonen kan også være resultat av if-tester i koden. Ved bakover bevisføring mister vi muligheten for å bruke forbetingelsen underveis i konstruksjonen. Vi må helt tilbake til forbetingelsen før vi kan bruke denne informasjonen til å forenkle predikatene.

4 Konklusjon

Bakover resonnering er etablert som en velegnet metode for å resonnere over tilordninger. Det er også dette som er utgangspunktet for flere nyere tilnærminger for å løse aliasproblemer som følge av objektorientering [3, 2]. I motsetning til å resonnere bakover over tilordninger, har vi i denne artikkelen resonnert forover. Vi har her presentert Hoare-regler samt en semantisk substitusjonsmekanisme som vi trenger for å sikre sunnhet. I tillegg har vi sett på forenklede aksiomskjemaer som brukes i resonnering. Vi har også argumentert for sunnhet av disse. Til sammen gir dette oss et system for å resonnere rundt programmer med pekere. De forenklede aksiomskjemaene vi har gitt, har syntaktisk kontrollerbare sidebetingelser, som sikrer sunnhet i resonneringen. Syntaktisk kontrollerbare sidebetingelser bidrar også til å bedre automatiserbarheten av systemet. Med utgangspunkt i at forbetingelsen til koden inneholder all aliasinformasjonen som er tilgjengelig i systemet, har vi til slutt har argumentert for at foroverresonnering ser ut til å være en attraktiv metode for resonnering i objektorienterte språk.

Videre arbeid

Bornat [3] går et steg lenger enn vi har gjort her. I tillegg til å skille tilordninger på bakgrunn av komponentnavn, går han videre og deler opp minnet i disjunkte deler med bakgrunn i datastrukturer. Reynolds [14] gjør en tilsvarende oppdeling, noe som bedrer lokaliteten i tilordninger. Det virker lovende at lokalitet i datastrukturer også kan innarbeides i systemet vi har presentert her. Calcagno m.fl. [4] arbeider videre på tilnærmingen til Bornat og gir en beskrivelse av semantikken til denne, samt utvider språket med aksiomer for opprettelse og sletting av objekter.

Berg [2] påpeker at komponenter i objekter ofte er skjermet fra omverdenen (innkapslet) og at bare metoder som er definert i klassens interface er synlige utenfra. På bakgrunn av dette utvikler Berg en ny formalisme basert på de samme grunntankene som Morris og Bornat, men der aliasingproblemer oppstår som følge av metodekall og ikke komponentoppdateringer. Dette muliggjør resonnering på et mer objektorientert plan. Berg får imidlertid de samme problemene hva angår størrelsene på predikatene ved tilbakeføring som Bornat har. I denne artikkelen har vi ikke hatt fokus på metodekall, men det kan være interessant å undersøke hvordan resultatene i denne artikkelen eventuelt lar seg benytte i resonnering rundt metoder.

For å få et fullt utviklet system er det videre nødvendig å utvide språket med subklasser. Pierik og de Boer [13] gjør en nylig tilnærming til dette, ved å innarbeide det samme aksiomskjemaet som Bornat i et språk med subklasser.

Takk

Takk til professor Olaf Owe for nyttige kommentarer og innspill under arbeidet med denne artikkelen. Han fortjener også en stor takk for god og konstruktiv veiledning under arbeidet med hovedoppgaven, som ligger til grunn for denne artikkelen.

Referanser

[1] Krzysztof R. Apt. Ten years of Hoare's logic: A survey–part i. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oktober 1981.

[2] Henrik Berg. Aliasproblematikk innen Hoare-logikk for objektorientert programmering. Hovedoppgave, Institutt for informatikk, Universitet i Oslo, November 2002.

[3] Richard Bornat. Proving pointer programs in Hoare Logic. I R. Backhouse og J. Nuno Oliveira, redaktører, *LNCS 1837, Proceedings of MPC 2000*, side 102–126. Springer-Verlag, 2000.

[4] Cristiano Calcagno, Samin Ishtiaq og Peter W.O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. I *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, side 190–201. ACM Press, 2000.

[5] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Computing*, 7(1):70–90, Februar 1978.

[6] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[7] Johan Dovland. Pekeraliasing i Hoare-logikk. Hovedoppgave, Institutt for informatikk, Universitet i Oslo, November 2002.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oktober 1969.

[9] Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[10] David C. Luckham og Norihisa Suzuki. Verification of array, record and pointer operations in pascal. *ACM Transactions on programming languages and systems*, 1(2):226–244, Oktober 1979.

[11] Joseph M. Morris. A general axiom of assigment. assignment and linked data structures. a proof of the schorr-waite algorithm. I Gunther Schmidt Manfred Broy, redaktør, *Theoretical Foundations of Programming Methodology*, side 25–51. Reidel, 1982.

[12] Olaf Owe. Partial Logics Reconsidered: A Conservative Approach. *Formal Aspects of Computing*, 5:208–223, 1993.

[13] Cees Pierik og Frank S. de Boer. A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts. I *Proceedings of FMOODS 2003*. Springer-Verlag. Under utgivelse.

[14] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. I Jim Davies, Bill Roscoe og Jim Woodcock, redaktører, *Millennial Perspectives in Computer Science*, side 303–321, Houndsmill, Hampshire, 2000. Palgrave. ISBN 0-333-92230-1.