



---

# Polymorf constraint-baseret typeinferens

*Featherweight Java og Featherweight Generic Java—elimination af down-casts*

Datalogisk Institut, Københavns Universitet

---

*Henrik Stuart*

*11. januar 2006*

## Resumé

Vi præsenterer Featherweight Java og Featherweight Generic Java og illustrerer de forskellige typer casts der findes i Featherweight Java, samt hvilke af disse der kan undgås i Featherweight Generic Java. Vi præsenterer dernæst Wang og Smiths framework for *constraint*-baseret polymorf objekt-orienteret typeinferens, og instantieringer af Agesens *cartesian product algorithm* og Wang og Smiths *data polymorphic cartesian product algorithm* til dette framework. Til sidst skitserer vi, hvordan man kan benytte de *constraint*-baserede typeinferens-algoritmer til at fjerne overflødige downcasts, som kun har nødet at eksistere i Featherweight Java.

– Henrik Stuart

# Indhold

Illustrationer	iii
1 Motivation	1
2 Grundkalkyler	2
2.1 Featherweight Java . . . . .	2
2.2 Featherweight Generic Java . . . . .	2
3 Casts	4
3.1 Overflødige downcasts . . . . .	4
3.2 Downcasts og manglende genericitet . . . . .	5
3.3 Uundgåelige downcasts . . . . .	5
4 Polymorf constraint-baseret typeinferens	7
4.1 Cartesian product algorithm . . . . .	11
4.2 Data polymorphic cartesian product algorithm . . . . .	12
5 Fremtidigt arbejde	13
6 Litteratur	14
A Typeinferens	15

## Illustrationer

1 Forskel på <i>erasure</i> - og type-bærende baserede implementationer .	3
2 Undgåelig downcast . . . . .	4
3 Downcast, der skyldes manglende genericitet . . . . .	6
4 Uundgåelig downcast . . . . .	8
5 Datapolymorft program . . . . .	10

### 1 Motivation

I C# skelner man mellem værdityper og referencetyper, som henholdsvis placeres på stakken og på hoben.<sup>1</sup> Når en instans af en værditype konverteres til en referencetype, skal den indkapsles og placeres på hoben, og den modsatte vej skal den pakkes ud og placeres på stakken. Dette kaldes for *boxing* henholdsvis *unboxing*. Disse konverteringer har en vis omkostning, så derfor vil det være en fordel at kunne minimere antallet af dem.

Der er to umiddelbare metoder til dette. Den ene vil typisk være at kreere en specialiseret *container*, der behandler en specifik værditype. Dette gøres typisk ved at arve fra `System.Collections.CollectionBase` og implementere en række metoder på denne klasse. Denne metode anbefales af Microsoft, men internt benytter `CollectionBase` en `ArrayList` instans, der arbejder på referencetyper, så vi får stadig *boxing* og *unboxing*. For at undgå dette, skal man selv implementere et dynamisk array for sine værdityper. Den anden metode er at se sit program som et C# 2 program og forsøge at inferere typer for de dele, hvor der er downcasts. Det vil for eksempel betyde, at vi vil forsøge at omforme en `ArrayList` instans, der kun indeholder instanser af `Int32`, til `List<Int32>`. Vi kigger her på fundamentet for sidstnævnte metode.

---

<sup>1</sup>Lignende overvejelser gør sig gældende for Java 1.4, men min primære motivation for at undersøge dette er for at få performance optimeringer i .NET.

## 2 Grundkalkyler

For lettere at kunne ræsonnere om Java og Generic Java, præsenterer Igarashi et al. (1999) Featherweight Java og Featherweight Generic Java som grundkalkyler for disse to sprog. Featherweight Java og Featherweight Generic Java er designet, så de repræsenterer den funktionelle kerne af Java og Generic Java, men uden at bevise for typesystemet kompliceres i nævneværdig grad.

Vi præsenterer kort Featherweight Java og Featherweight Generic Java i de næste to afsnit.

### 2.1 Featherweight Java

Pierce (2002, kapitel 19) og Igarashi et al. (1999) præsenterer Featherweight Java, men da Pierce (2002, kapitel 19) stort set er en gentagelse af Igarashi et al. (1999), vil vi fokusere på sidstnævnte.

Featherweight Java indeholder objektkonstruktion, metodekald, felttilgang, variable og casts ud over klasseerklæringer med dertil hørende felter, metoder og konstruktør. For simplificeringens skyld, er der ikke tildelingsudtryk i Featherweight Java, så Igarashi et al. antager, at objektet altid bliver initialiseret i dens konstruktør. Sproget repræsenterer en funktionel kerne af Java, og er konstrueret således, at det er indeholdt i Java. Derfor kan ethvert Featherweight Java program oversættes og afvikles som et Java program.

Da Featherweight Java repræsenterer den funktionelle kerne af Java, er der ikke sideeffekter i sproget, så metoder er begrænsede til blot at være return efterfulgt af et udtryk.

Syntaks, evaluering og typning af Featherweight Java er vist i Igarashi et al. (1999, figur 1) og hjælpefunktionerne til denne i Igarashi et al. (1999, figur 2). Der er ingen overraskelser i forhold til Java.

### 2.2 Featherweight Generic Java

Featherweight Generic Java er en udvidelse af Featherweight Java, der skal repræ-

## Typer og programmeringssprog

```
Object o = new List<C>();  
List<D> d = (List<D>)o;
```

Illustration 1: Forskel på *erasure*- og type-bærende baserede implementationer

sentere Generic Java, men ulig Featherweight Java, så er Featherweight Generic Java ikke indeholdt i Generic Java, da Igarashi et al. for simplifikationens skyld har undladt typeinferens for generiske metoder.

Syntaks, evaluering, subtypning og typning af Featherweight Generic Java er vist i Igarashi et al. (1999, figur 3 og 4), og derudover er diverse hjælpefunktioner til disse vist i Igarashi et al. (1999, figur 5). Syntaksen minder i det store hele om Generic Java, bortset fra, at alle generiske parametre, som nævnt, skrives fuldt ud.

Som Igarashi et al. skriver, er det meningen, at Featherweight Generic Java enten implementeres ved en type-bærende implementation, hvor typerne bliver båret med rundt på kørselstidspunktet, eller en *erasure*-baseret implementation, hvor programmet oversættes til et ikke-generisk sprog.<sup>2</sup> Forskellen mellem implementationerne bliver vigtig, når vi arbejder med downcasts, såfremt man benytter en naiv udvidelse af downcasts fra Featherweight Java. Hvis vi eksempelvis benytter en *erasure*-baseret implementation på programmet i illustration 1, vil downcastet lykkes, da enhver liste bliver omformet til en liste af objekter, mens i en type-bærende implementation ville evalueringen blive *stuck*. Derfor bliver downcasts i Featherweight Generic Java og Generic Java specificeret, så de kun giver mening, hvis de er gyldige i begge implementationer.

---

<sup>2</sup>For vores motivation omkring C#, vil det sidstnævnte betyde, at vi ville få præcis de samme problemer med *boxing* og *unboxing*.

## Typer og programmeringssprog

```
class A extends Object {
    A() { super(); }
    A Append(B b) { return this; }
}

class B extends Object {
    B() { super(); }
}

class C extends B {
    C() { super(); }
    B AsB() { return (B)this; }
}
```

Illustration 2: Undgåelig downcast

### 3 Casts

Der er tre typer casts i Featherweight Java og Featherweight Generic Java: upcasts, downcasts og det Igarashi et al. kalder stupid casts. Vi fokuserer udelukkende på downcasts i dette afsnit. Der er tre kategorier af downcasts: overflødige downcasts; downcasts, der er forårsaget af manglende genericitet; og downcasts, der ikke kan undgås i Featherweight Generic Java.

#### 3.1 Overflødige downcasts

Vi kalder et downcast overflødigt, hvis det ikke er nødvendigt, for at et program er typekorrekt.

I programstumpen i illustration 2, sammenholdt med det nedenstående udtryk, og da Append forventer en instans af typen B og vi allerede har en instans af type B, er downcastet (C) overflødigt.

```
(new A()).Append((C)(new C()).AsB())
```

### 3.2 Downcasts og manglende genericitet

Vi kigger i dette afsnit på downcasts, der kan fjernes ved at oversætte et program fra Featherweight Java til Featherweight Generic Java og tilføje typeabstraktioner. Vi illustrerer denne type downcasts ved hjælp af `Pair` eksemplerne fra Igarashi et al., som er gengivet i illustration 3 på den følgende side.

Hvis vi ønsker at skabe et par og hente det første element ud med dets mest afledte type, ville vi være nødt til at downcaste, da `Pair` arbejder på instanser af typen `Object` i Featherweight Java. Derimod vil vi i Featherweight Generic Java være i stand til at specificere, hvilke typer `Pair` arbejder på. Dette betyder forskellen mellem

```
(A)(new Pair(new A(), new A())).fst
```

og

```
(new Pair<A,A>(new A(), new A())).fst
```

Ved at omskrive programmet fra Featherweight Java til Featherweight Generic Java er vi i stand til at fjerne et downcast, og dermed en yderligere fejlkilde, da der ikke længere kan være en potentiel kørselstidsfejl. Det er disse downcasts vi er interesserede i at fjerne automatisk på længere sigt.

### 3.3 Uundgåelige downcasts

Den sidste type downcasts kan ikke undgås i Featherweight Generic Java. Denne type downcasts vil som regel forekomme i en *container*, der accepterer instanser af heterogene typer, altså klasser der kun har `Object` som fælles forælder.

Det nedenstående udtryk evalueret over programmet i illustration 4 på side 8 kan ikke undgås, da vi ikke kan abstrahere homogent over typerne af de lagrede elementer, det vil sige, at `Object` vil være den nedre grænse for de lagrede instansers typer. Såfremt vi kunne typeabstrahere klassevariable, som i for eksempel

## Typer og programmeringssprog

```
// Featherweight Java
class A extends Object {
    A() { super(); }
}

class Pair extends Object {
    Object fst;
    Object snd;

    Pair(Object fst, Object snd) {
        super();
        this.fst = fst;
        this.snd = snd;
    }
}

// Featherweight Generic Java
class A extends Object {
    A() { super(); }
}

class Pair<T extends Object, U extends Object> extends Object {
    T fst;
    U snd;

    Pair(T fst, U snd) {
        super();
        this.fst = fst;
        this.snd = snd;
    }
}
```

Illustration 3: Downcast, der skyldes manglende genericitet

C++, ville vi være i stand til at ændre programmet, så nedenstående udtryk kan udføres uden downcast.

```
(A)(new List(new A(), new Nil())).Cons(new B()).Tail().hd
```

### 4 Polymorf constraint-baseret typeinferens

Vi kigger i dette afsnit på Wang og Smiths framework for objekt-orienteret polymorf constraint-baseret typeinferens. Selvom det sprog de benytter påstås at være Featherweight Java udvidet med felttildeling, så er der en række afvigelser.

- Metoder er blot et udtryk—ikke return efterfulgt af et udtryk. Udtrykket definerer dermed returtypen for metoden.
- Klasser har ikke længere en konstruktør. Klasseinstantieringer foretages udelukkende som `new  $\delta$` , hvor  $\delta$  er klassenavnet.
- Der er tilføjet sekvensoperator  $(e_1; e_2)$ , hvor klassedefinitioner fra  $e_1$  kan anvendes i  $e_2$ , og  $e_2$  afgør typen af udtrykket.
- Der er ikke længere nogle typeannotationer.

Desuden bryder Wang og Smith med Igarashi et al., da et program ikke længere er en tupel af klassedefinitioner og et udtryk. Et program i Wang og Smiths sprog er blot et udtryk. Det vil sige, at for at kunne benytte en klassedefinition må man opstille den som en sekvens af operationer. Dette betyder blandt andet, at klasser ikke kan benytte hinanden cyklisk, som de ellers kan i Java.

Vi benytter i det følgende sprogdefinitionen i Wang og Smith, definition 5.1, og fokuserer på at forklare deres framework. Vi bemærker kun Wang og Smiths afvigelser fra deres sprog, hvis det berører vores gennemgang af frameworket.

Frameworket forsøger at indfange informationsflowet i et program, og Wang og Smith har brugt dette til at beregne, hvorvidt downcasts er sikre eller usikre—

## Typer og programmeringssprog

```
class List extends Object {
  Object hd;
  Object tl;

  List(Object hd, Object tl) {
    super(); this.hd = hd; this.tl = tl;
  }

  List Cons(Object hd) {
    return new List(hd, this);
  }

  List Tail() {
    return (List)tl;
  }
}

class Nil extends Object {
  Nil() { super(); }
}

class A extends Object {
  A() { super(); }
}

class B extends Object {
  B() { super(); }
}
```

Illustration 4: Uundgåelig downcast

## Typer og programmeringssprog

det vil sige, hvorvidt man statistisk kan udlede om et downcast er gyldigt på kørselstidspunktet eller ej. Den måde frameworket indfanger informationsflowet på, er ved først at generere en *constraint* mængde ud fra programmet (udtrykket) efter reglerne i Wang og Smith, figur 3, og dernæst beregne lukningen (eng. *closure*) af denne. Udregningen af lukningen foretages ved brug af reglerne i Wang og Smith, figur 4.

Wang og Smith har i deres framework gjort det muligt at analysere metodekald og klasseinstantieringer efter forskellige algoritmer. Dette er gjort i  $\forall$ -Elim reglen. Vi kigger i de næste afsnit på to algoritmer: Agesens *cartesian product algorithm* (CPA) og Wang og Smiths *data polymorphic cartesian product algorithm* (DCPA). En algoritmes analyse bliver betegnet en *contour*, og det er op til den enkelte algoritme at bedømme, hvorvidt en genereret *contour* kan deles mellem flere kald og instantieringer.

Datapolymorfi forekommer, når en imperativ variabel kan tildeles instanser af forskellige typer på kørselstidspunktet. Det program Wang og Smith benytter til at illustrere problematikken omkring datapolymorfi, figur 5 i Wang og Smith, er ikke gyldig i forhold til deres sprog, men er udtrykt direkte i Java. Vi har i illustration 5 på den følgende side forsøgt at opstille et eksempel i deres sprog, som vi kan benytte til at illustrere *constraint* genereringen, og hvordan CPA og DCPA virker. Vi antager, at `Object` er erklæret udenfor programmet, og dermed ikke skal erklæres i hvert program.

Hvis vi kigger på typeinferensen for vores eksempel, så er inferensalgoritmen fra Wang og Smith, figur 3, implementeret ad hoc—det vil sige, det ikke er alle inferensreglerne, der er implementeret fuldstændigt men nok til at inferere typer i vores eksempel. Programmet kan ses i appendix A. Den *constraint* mængde der bliver genereret er som følger.<sup>3</sup>

---

<sup>3</sup>Hvis ellers programmet er implementeret korrekt. At beregne det i hånden viste sig for besværligt.

## Typer og programmeringssprog

```
class A extends Object { };
class B extends Object { };

class Dynamic extends Object {
    value;

    Assign(o) this.value = o; this

    Read() this.value
};

(A)(new Dynamic).Assign(new A).Read();

(B)(new Dynamic).Assign(new B).Read()
```

### Illustration 5: Datapolyomorft program

$$\begin{aligned} & \{(\forall\{t_1\}.t_1 \rightarrow \mathbf{obj}(A, [])) \setminus \{\}\} <: t_A, \\ & (\forall\{t_2\}.t_2 \rightarrow \mathbf{obj}(B, [])) \setminus \{\} <: t_B, \\ & (\forall\{t_7\}.t_7 \rightarrow \mathbf{obj}(Dynamic, [field : t_8, \\ & \quad Assign : (\forall\{t_3, t_4\}.(t_4, t_3) \rightarrow t_3) \setminus \{t_3 <: [value : \mathbf{write} t_4]\}, \\ & \quad Read : (\forall\{t_5, t_6\}.t_5 \rightarrow t_6) \setminus \{t_5 <: [value : \mathbf{read} t_6]\}]) \setminus \{\} <: t_{Dynamic}, \end{aligned}$$

## Typer og programmeringssprog

$$\begin{array}{ll}
 t_{16} <: [\text{Read} : t_{17} \rightarrow t_{16}], & \mathbf{null} <: t_{12}, \\
 t_A <: t_{12} \rightarrow t_{11}, & \mathbf{null} <: t_{10}, \\
 t_{\text{Dynamic}} <: t_{10} \rightarrow t_9, & \mathbf{null} <: t_{22}, \\
 \mathbf{null} <: t_{20}, & t_{21} <: t_{23}, \\
 t_{26} <: \mathbf{cast}(B, t_{28}), & t_{11} <: t_{13}, \\
 t_{14} <: [\text{Assign} : (t_{13}, t_{15}) \rightarrow t_{14}], & t_{16} <: \mathbf{cast}(A, t_{18}), \\
 t_{26} <: [\text{Read} : t_{27} \rightarrow t_{26}], & t_B <: t_{22} \rightarrow t_{21}, \\
 t_{\text{Dynamic}} <: t_{20} \rightarrow t_{19}, & t_{24} <: [\text{Assign} : (t_{23}, t_{25}) \rightarrow t_{24}] \}
 \end{array}$$

Genereringen af *constraints* forløber forholdsvis problemfrit, men reglen for metodekald er lettere uklar, da det ikke er oplagt, hvorvidt der bliver genereret nye typer for  $(\bar{t}_i \times t') \rightarrow t$ , eller om de bliver genbrugt fra metodefinitionen. Vi har valgt at generere nye typer, da man, så vidt vi kan se, ikke har den specifikke klasstype, som metoden er erklæret i, på dette tidspunkt.

På grund af tidsbegrænsninger, har vi ikke implementeret *closure* beregningen, men vil nøjes med i de næste to afsnit at illustrere problematikken omkring datapolymorfi, samt overordnet, hvordan henholdsvis CPA og DCPA analyserer programmet i illustration 5 på foregående side.

### 4.1 Cartesian product algorithm

CPA bliver præsenteret i Agesen (1995), og en instantiering til Wang og Smiths framework for objekt-orienteret programanalyse bliver kort beskrevet i Wang og Smith, afsnit 5.2. CPA bliver kaldt ved analyse af klasseinstantieringer og metodekald, men for nemheds skyld skriver vi udelukkende metodekald i det følgende, da klasseinstantieringen, som Wang og Smith skriver, er et specialtilfælde af metodekald.

CPA analyserer kaldstedet, det vil sige et metodekald, ud fra de argumenter der anvendes, og genererer *constraints* for hvordan informationen propagerer

## Typer og programmeringssprog

gennem metoden. For hver kombination af argumenter metoden anvendes med, bliver der genereret en *contour*. To kald deler samme *contour*, hvis og kun hvis argumenttyperne i de to kald er identiske.

Hvis vi benytter CPA som algoritme for vores eksempelprogram i illustration 5 på side 10, betyder det, da vi tildeler `Dynamic::value` med henholdsvis en værdi af type  $t_A$  og  $t_B$ , at returmængden for `Read`, der læser denne instansvariabel, er  $\{t_A, t_B\}$ , mens returtypen for `Assign`, der altid returnerer `this`, vil være  $\{t_{Dynamic}\}$ . `Read` illustrerer problemet med datapolymorfi, da CPA deler *contour*'en for `Read` mellem begge udtryk. Vi er derfor ikke i stand til at skelne mellem, hvornår instansen indeholder en værdi af type  $t_A$  eller  $t_B$ , selvom vi kan se på programmet, at begge downcasts vil lykkes. Da både  $t_A$  og  $t_B$  propagerer til `Dynamic::value` og deres fælles nedre grænse er  $t_{Object}$ , må `Dynamic::value` nødvendigvis have type  $t_{Object}$ , og det er her vi klart ser datapolymorfien.

### 4.2 Data polymorphic cartesian product algorithm

Som vi har vist i forrige afsnit, så er datapolymorfi skyld i, at CPA tilknytter samme *contour* til en metode for flere kald, selvom det ikke er hensigtsmæssigt. Dette problem forsøger Wang og Smith at løse ved at udbygge CPA, så den kan bedømme hvornår den skal genbruge *contours* eller ej. Den nye algoritme de præsenterer er DCPA.

DCPA benytter CPA ved lukningsberegningen, men efter kørsel af CPA bedømmer DCPA, hvorvidt den genererede *contour* bruges i forbindelse med datapolymorfi. Såfremt den genererede *contour* ikke mister præcision, altså ikke bliver brugt datapolymorft, er den CPA-*safe*, ellers er den CPA-*unsafe*. Såfremt DCPA skønner, at en *contour* er *unsafe*, genererer den en ny *contour* ved følgende anvendelser. Vi ser nedenfor på, hvordan DCPA bedømmer *contours* i hvert tilfælde.

Klasseinstantiering: For en klasse  $A$ , vil dens instantieringsskema,  $(\forall \vec{t}. \tau \setminus C)$ , blive beregnet som CPA-*unsafe*, hvis et eller flere af dens fel-

## Typer og programmeringssprog

ter er datapolymorfe. Ellers vil instantieringsskemaet blive beregnet som *safe*.

Metodekald: Hvis instanser af en klasse har datapolymorfe dele, forsøger DCPA at generere flere *contours* for de enkelte anvendelser af disse.<sup>4</sup>

Såfremt en metode ikke instantierer en klasse, eller såfremt en metode ikke returnerer en instans af en klasse, da beregnes DCPA som *safe*. Desuden er en metode der returnerer en instans af en klasse *safe*, såfremt den er instantieret med en primitiv type—i vores sprog vil dette være `int`. Alle andre kald, hvis vi ignorerer deres udvidede heuristik, vil blive beregnet *unsafe*.

Selve DCPA algoritmen, uden heuristik, udvidet med de resterende *closure* beregninger, er vist i Wang og Smith, figur 7.

Hvis vi benytter DCPA på vores program i illustration 5 på side 10, betyder det, at de to instantieringer af `Dynamic` vil danne separate *contours*. Vi har stadig, at både  $t_A$  og  $t_B$  er nedre grænser for `value`, som derfor må have type  $t_{Object}$ . Dog vil DCPAs analyse vise, at `value` ikke kan indeholde både  $t_A$  og  $t_B$  i hvert af de to udtryk, og returtypemængden for `Read` bliver derfor  $\{t_A\}$  i det første udtryk, henholdsvis  $\{t_B\}$  i det andet udtryk, og dermed kan vi udlede, at `downcast` vil lykkes på kørselstidspunktet i begge tilfælde.

## 5 Fremtidigt arbejde

Da Wang og Smiths framework kan benyttes til at bedømme, hvorvidt et `downcast` er *safe* eller *unsafe*, burde det være muligt at tilpasse det til at styre en oversæt-

---

<sup>4</sup>DCPA indeholder i Wang og Smith også en heuristik, der begrænser antallet af *contours* der genereres. Vi vil se bort fra denne udvidelse her, da den ikke er vigtig i forhold til hvordan DCPA konceptionelt virker.

telse fra Featherweight Java til Featherweight Generic Java ved at finde passende polymorfe udgaver for de steder, hvor downcasts kan isoleres som værende *safe*. At gøre dette, falder desværre udenfor omfanget af denne opgave.

Muligheden for dette kan, hvis vi benytter DCPA som instantiering til frameworket, ses i illustration 5 på side 10, hvor klassen `Dynamic` kan typeabstraheres over en type `T`, og vi kan lade `value` antage typen `T`. Da kan vi instantiere `Dynamic` med henholdsvis `A` og `B` i de to udtryk, og vi vil ikke længere behøve et downcast. På baggrund af denne motivation, håber vi på, at man kan formalisere, hvornår sikre downcasts kan foranledige en typeabstraktion af en klasse/metode.

## 6 Litteratur

Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. I *ECOOP*, side 2–26, 1995.

Atshushi Igarashi, Benjamin Pierce, og Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. I Loren Meissner, redaktør, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, bind 34(10), side 132–146, New York, 1999.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

Tiejun Wang og Scott F. Smith. Polymorphic constraint-based type inference for java. Ikke udgivet.

## A Typeinferens

```

fun setnew () = [];
fun isin e xs = foldr (fn (x,v) => if v then v else e = x) false xs;

fun setadd e s = if isin e s then s else e::s;
fun setminus e [] = []
  | setminus e (x::xs) = if e = x then xs else x::(setminus e xs);
fun setin e s = isin e s;
fun makeset xs =
  let
    fun lmakeset s [] = s
      | lmakeset s (x::xs) = (setadd x s) @ (lmakeset s xs);
  in
    lmakeset (setnew()) xs
  end;
fun setunion s1 [] = s1
  | setunion s1 (s::s2) = setunion (setadd s s1) s2;
fun setdiff s1 [] = s1
  | setdiff s1 (s::s2) = setdiff (setminus s s1) s2;

fun filter f [] = []
  | filter f (x::xs) = if (f x) then x::(filter f xs) else (filter f xs);

datatype Exp = new of string
  | read of Exp * string
  | write of Exp * string * Exp
  | methodcall of Exp * string * Exp list
  | this
  | cast of string * Exp
  | Object
  | var of string
  | class of string * string * string list * (string * (string * string) list * Exp) list
  | seq of Exp * Exp
  | method of string * (string * string) list * Exp;

datatype Type = tvar of string
  | all of Type list * Type * Constraint list
  | arrow of Type list * Type
  | obj of string * (string * Type) list
  | null
  | label of string * Type
  | writeT of Type
  | readT of Type
  | castT of string * Type
and
  Constraint = subt of Type * Type;

exception Missing;

fun writeType t =
  let
    fun writeList l =
      foldr (fn (x,e) => (writeType x) ^ (if e = "" then "" else ",") ^ e) "" l;
    fun writeObjList l =

```

## Typer og programmeringssprog

```
foldr (fn ((n,t),e) => n ^ " : " ^ (writeType t) ^ (if e = "" then "" else ",") ^ e) "" 1;
  fun writeTypeH (tvar t) = t
| writeTypeH (arrow (params, result)) =
"(" ^ (writeList params) ^ ") -> " ^ (writeType result)
| writeTypeH (all (quant, body, c)) =
"(All {" ^ (writeList quant) ^ "}. " ^ (writeType body) ^ "\\{" ^ (writeConstraints c) ^ "})"
| writeTypeH (null) = "null"
| writeTypeH (obj (d,f)) = "(" ^ d ^ ", [" ^ (writeObjList f) ^ "]"
| writeTypeH (castT (c,d)) = "cast(" ^ c ^ ", " ^ (writeType d) ^ ")"
| writeTypeH (label (c,d)) = "[" ^ c ^ " : " ^ (writeType d) ^ "]"
| writeTypeH (writeT x) = "write " ^ (writeType x)
| writeTypeH (readT x) = "read " ^ (writeType x)
  in
writeTypeH t
  end
and
  writeConstraint (subt (t1, t2)) = (writeType t1) ^ " <: " ^ (writeType t2)
and
  writeConstraints [] = ""
  | writeConstraints (x::xs) = (writeConstraint x) ^ "\n" ^ (writeConstraints xs);

local
  val counter = ref 0;
in
  fun newtvar () = (counter := !counter + 1; "t" ^ Int.toString(!counter));
end;

fun FindClass _ [] = raise Missing
  | FindClass name ((n,all x)::xs) = if name = n then (n,all x) else FindClass name xs
  | FindClass name ((n,_)::xs) = if name = n then raise Missing else FindClass name xs;

fun GetTvar (tvar s) = s
  | GetTvar _ = raise Missing;

fun FTVh (tvar s) = true
  | FTVh _ = false;

fun FTV (tvar s) bound = if setin (tvar s) bound then [] else [tvar s]
  | FTV (all (ss, t, c)) bound = (FTV t (bound @ (filter FTVh ss)))
  | FTV (arrow (ss, t)) bound = makeset ((filter FTVh ss) @ (FTV t bound))
  | FTV (obj (s,f)) bound = []
and FTVC (subt (t1,t2)) bound = (FTV t1 bound) @ (FTV t2 bound)
and FTVE [] bound = []
  | FTVE ((n,x)::xs) bound = (FTV x bound) @ (FTVE xs bound);

fun InferObject env =
  let
    val t = tvar (newtvar())
  in
    (env, Object, tvar "t_Object",
     [subt(all([t], arrow([t], obj("Object", [])), []), [], tvar "t_Object")])
  end;

fun GetClassName (tvar s) = implode (tl (tl (explode s)))
  | GetClassName _ = raise Missing;
```

## Typer og programmeringsprog

```
fun GetClasses [] Ts = []
  | GetClasses ((subt (all (z1,z2,z3), tvar y))::xs) Ts =
  if setin y Ts
  then (GetClassName (tvar y), all (z1,z2,z3))::(GetClasses xs Ts)
  else (GetClasses xs Ts)
  | GetClasses (_::xs) Ts = GetClasses xs Ts;

fun InferNew env s Ts =
  let
    val hasT = length (filter (fn (x,y) => x = s) env) > 0;
    val tv = tvar (newtvar());
    val tvz = tvar (newtvar());
    val classt = tvar ("t_" ^ s);
  in
    if hasT then
      (env, new s, tv, [subt(classt, arrow ([tvz], tv)), subt(null, tvz)])
    else
      raise Missing
  end;

fun Infer env (class (d, d', fields, methods)) Ts =
  let
    val (n, all x) = FindClass d' env;
    val inferM = map (fn x => Infer env (method x) Ts) methods;

    val tv = tvar (newtvar());

    val fieldM = map (fn x => (x, tvar (newtvar()))) fields;
    val methM = map (fn (_,method (n,_,_),x,_) => (n,x)) inferM;
    val memM = fieldM @ methM;
    val tmap = arrow ([tv], obj(d, memM));

    val quant = FIV tmap []
    val tt = all (quant, tmap, [])
    val classT = tvar ("t_" ^ d);
  in
    (env, class (d, d', fields, methods), classT, [subt(tt, classT)])
  end
  | Infer env (Object) Ts = InferObject env
  | Infer env (new s) Ts = InferNew env s Ts
  | Infer env (read (e1, field)) Ts =
  let
    val (env1, exp1, t1, c1) = Infer env e1 Ts;
    val t = tvar (newtvar());
  in
    (env, read (e1, field), t, [subt(t1, label(field, readT t))])
  end
  | Infer env (write (e1, field, e2)) Ts =
  let
    val (env1, exp1, t1, c1) = Infer env e1 Ts;
    val (env2, exp2, t2, c2) = Infer env e2 Ts;
  in
    (env, write (e1, field, e2), t2, [subt(t1, label(field, writeT t2))] @ c1 @ c2)
  end
  | Infer env (method (name, params, body)) Ts =
  let
```

## Typer og programmeringsprog

```
    val thist = tvar (newtvar());
    val nparamtype = map (fn (x,y) => (x, tvar (newtvar()))) params;
    val (env1, e1, t, c) = Infer (env @ [{"this", thist}] @ nparamtype)
body Ts
    val paramTypes = (map (fn (x,y) => y) nparamtype) @ [thist];
    val bodyT = arrow(paramTypes, t);
    val quant = setdiff (setdiff (FTV bodyT []) (FTVE env [])) (map (fn x => tvar x) Ts);
    val _ = print (writeConstraints c);
in
    (env, method (name, params, body), all(quant, bodyT, c), [])
end
| Infer env (var s) Ts =
let
    val t = filter (fn (x,y) => x = s) env;
    val tt = if length t > 0 then #2 (hd t) else raise Missing;
in
    (env, this, tvar ((fn (tvar q) => q) tt), [])
end
| Infer env this Ts = Infer env (var "this") Ts
| Infer env (cast (c,e)) Ts =
let
    val (env1,exp1,t1,c1) = Infer env e Ts;
    val t = tvar (newtvar());
in
    (env, cast (c,e), t, setunion [subt(t1, castT (c, t))] c1)
end
| Infer env (methodcall (e1, m, args)) Ts =
let
    val (env1,exp1,t1,c1) = Infer env e1 Ts;
    val argsInfer = map (fn x => Infer env x Ts) args;
    val argsTypes = (map (fn (_,_,x) => x) argsInfer);
    val argsConstraints = (map (fn (_,_,x) => x) argsInfer);
    val argsOrigTypes = (map (fn _ => tvar (newtvar())) argsTypes);
    fun makeSubt [] [] = []
| makeSubt (x::xs) (y::ys) = (subt (x,y)) :: (makeSubt xs ys);
    val argsTC = makeSubt argsTypes argsOrigTypes;
    val t = tvar (newtvar());
    val mlabel = label(m, arrow(argsOrigTypes @ [tvar (newtvar())], t));
    val argsC = foldr (fn (x,y) => setunion x y) [] argsConstraints;
in
    (env, methodcall (e1, m, args), t,
    setunion (setunion (setunion [subt(t, mlabel)] argsTC) c1) argsC)
end
| Infer env (seq (e1,e2)) Ts =
let
    val (env1,exp1,t1,c1) = Infer env e1 Ts;
    val newclasses = GetClasses c1 Ts;
    val (env2,exp2,t2,c2) = Infer (newclasses @ env) e2 Ts;
in
    (env, seq (e1,e2), t2, c1 @ c2)
end;

val initEnv = [{"Object", (all([tvar "t_0", arrow([tvar "t_0", obj("Object", [])], [])])]);

val myA = class ("A", "Object", [], []);
```

## Typer og programmeringsprog

```
val myB = class ("B", "Object", [], []);
val myAssignInner = ("Assign", [{"o", "t_Object"}], seq(write (this, "value", var "o"), this))
val myAssign = method myAssignInner;
val myReadInner = ("Read", [], read (this, "value"))
val myRead = method myReadInner;
val programClasses = ["t_A", "t_Object", "t_B", "t_Dynamic"];
val myNew = new ("A");
val myDynamic = class ("Dynamic", "Object", ["field"], [myAssignInner, myReadInner]);
val myE1 = cast ("A", methodcall (methodcall (new "Dynamic", "Assign", [new "A"]), "Read", []));
val myE2 = cast ("B", methodcall (methodcall (new "Dynamic", "Assign", [new "B"]), "Read", []));

val a = Infer initEnv (seq (myA, seq (myB, seq (myDynamic, seq (myE1, myE2)))) programClasses;

print (writeConstraints (#4 a));
```