Fortgeschrittene Objektorientierung

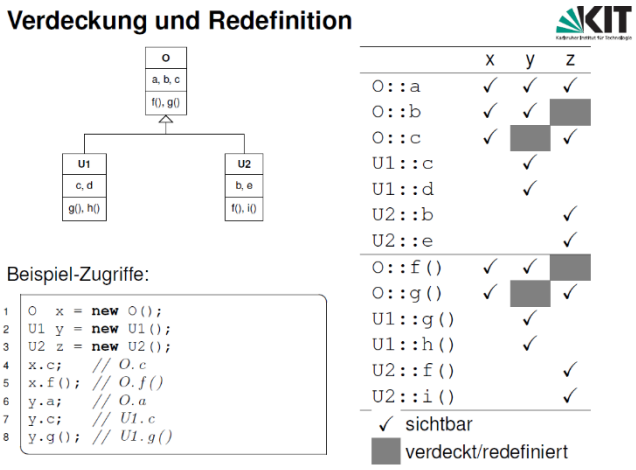
Lukas Bach - lbach@outlook.de - lukasbach.com

1 Einfachvererbung

1.1 Dynamische Bindung

Dynamische Bindung entscheidet zur Laufzeit anhand des Objekttyps, welche Methode aufgerufen wird.

Der konkreteste Typ wird verwendet, auf Oberklassen-Implementierungen kann mit `super.f()` zugegriffen werden (`super` schaltet dynamische Bindung ab).



- Statische Methoden werden immer statisch gebunden
- Private Methoden werden immer statisch gebunden.

1.1.1 Upcasts

```
U1 x = new U1();
```

- `((O) x).c` greift auf `O::c` zu. *Attribute* sind *statisch* gebunden.
- `((O) x).f()` greift auf `U1::f()` zu. *Methoden* sind *dynamisch* gebunden.
  - ⇨ Upcasts schalten dynamische Bindung nicht ab!

1.1.2 Objektlayout und Subobjekte

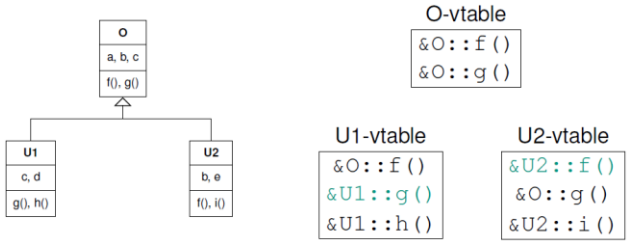
Im Objektlayout kommen zuerst geerbte Attribute, dann eigene. Zugriff erfolgt über statische Offsets.

Statischer Typ bestimmt verwendeten Offset, Statische Typisierung verhindert Zugriff mit illegalen Offsets, Dynamischer Typ der referemzierten Objekte ist irrelevant (Statischer Typ garantiert Sicherheit).

1.2 Methodentabelle (vtable)

Methodentabellen pro Klassen enthalten Einsprungadressen der Methoden. Ein Eintrag pro Methode inkl. geerbte Methoden.

Redefinierte Methoden haben immer dieselbe Position in der Methodentabelle.



Jedes Objekt enthält einen Zeiger (vptr) auf zugehörige Methodentabelle.

Dynamische Bindung in konstanter Zeit, da: Compiler kennt statischen Offset des `vptr`'s, kennt Tabellenindex von `g()` in der Methodentabelle und kann `g()` in konstanter Zeit von geladener Einsprungadresse aufrufen.

Realisierung this-pointer:

```
struct C {
  vtable *vptr;
  int x;
};
void m(C *this, int y) {
  *(this + offset(C::x)) = y;
  (*(*(this + offset(vptr))
    + offset(C::m)))(this, y);
}
```

Achtung: Methodenaufrufe über this unterliegen dynamischer Bindung und gehen an konkretesten Typen! Kann über `super` abgeschaltet werden.

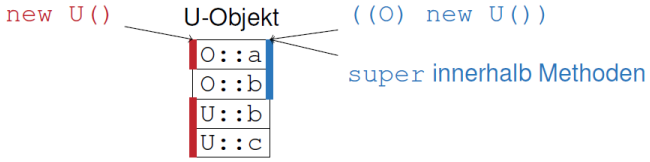
Private methoden sind wegen Geheimnisprinzip immer statisch gebunden.

1.3 Type Casts

1.3.1 Upcast

```
Oberklasse x = new Unterklasse();
```

ist ein impliziter Upcast, verdeckte Oberklassenattribute werden wieder sichtbar, aber Methoden nicht.



1.3.2 Downcast

```
Oberklasse x;
((Unterklasse) x).unterklassenAttr = 1;
```

Funktioniert nur, wenn x mit einer Instanz der Unterklasse instantiiert wurde, also **STATISCH UNSICHER** und **NUR ZUR LAUFZEIT ÜBERPRÜFBAR**.

1.3.3 Statische & dynamische Variablenbindung

TODO

2 Softwaretechnische Aspekte

2.1 OO vs imperative Programmierung

IMPERATIVER ANSATZ: zB Structs die Datentypen beschreiben, Unions zur Zusammenfassung der Structs und Methoden auf den Unions die nach Structtyp unterscheiden.

```
void move(union Shape s) {
  if (s.type == CIRCLE) {
    ...
} else if (s.type == RECT) {
    ...
} }
```

Verletzt *Geheimnisprinzip* und *Lokalitätsprinzip*.

OBJEKTORIENTIERTER ANSATZ: Abstrakte Klasse mit erbenden Unterklassen. *Geheimnisprinzip* und *Lokalitätsprinzip* gewahrt. Aber *Tyrannei der dominanten Dekomposition*, bei neuer Funktion müssen alle Klassen geändert werden.

3 Tücken der dynamischen Bindung

```
class P {
  void m() { m() } }
class C extends P {
  @Override void m() { super.m() } }
```

P.m ruft m aus C aus, nicht sein eigenes (dynamische Bindung greift auch bei Rekursion).

Lösungsansätze:

- `this.m()` funktioniert nicht, da `this` auf `C` zeigt
- `((P) this).m()` funktioniert nicht, Upcasts schalten dynamische Bindung nicht ab
- `P::m()` funktioniert in C++.

Dynamische Bindung geschieht auch im Konstruktor und bei Delegation.

4 Invarianten und sichere Vererbung

Verhaltenskonformanz: Aus Sicht des Methodenverhaltens ist jedes Unterklassenobjekt auch ein Oberklassenobjekt. Oberkl.-Obj, kann garantiert durch Unterkl.-Obj. ersetzt werden.

VERHALTENSKONFORMANZ: Für Klasseninvarianten (gelten vor und nach jedem Methodenaufruf) gilt:

Unterkl.-Invariante gilt stärker als Oberkl.-Invariante: $Inv(U) \Rightarrow Inv(O)$

Oberklasse hat stärkere Vorbedingung: $PRE(O.m) \Rightarrow PRE(U.m)$ (Oberklasse verlangt mehr).

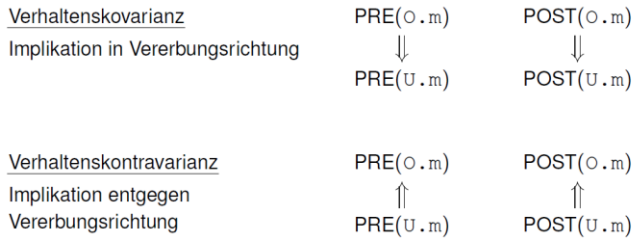
Unterklasse hat stärkere Nachbedingung: $POST(U.m) \Rightarrow POST(O.m)$ (Unterklasse leistet mehr).

Tipp: Bei Prüfung Bedingungen komplett ausschreiben.

Beispiel siehe F57 und F63.

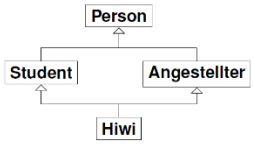
SPEZIALISIERUNG: Verhaltenskonformanz in der Praxis selten, stattdessen Spezialisierung: Unterklasse verlangt mehr, bietet funktional weniger. zB Implementierung für Spezialfälle (effizientere LinkedList von List, identische Nachbedingungen).

Effizienter, kann dafür abstürzen.



Inheritance	Subtyping
Wiederverwendung von Methodenimplementierungen	Wiederverwendung von Klientencode
Spezialisierung	Verhaltenskonformanz
+ Billige Klassenimpl.	- teure Unterklassen
- Teurer beim Klient (Absturz)	+ Billiger Klient
	+ Lokalitätsprinzip für neue Unterklassen

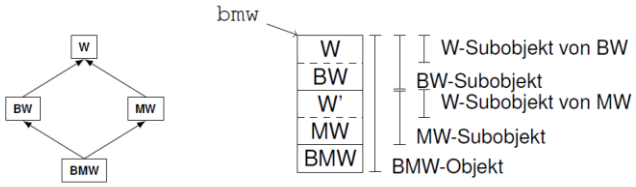
5 Mehrfachvererbung



Möglich in C++.

5.1 Nicht-virtuelle Vererbung in C++

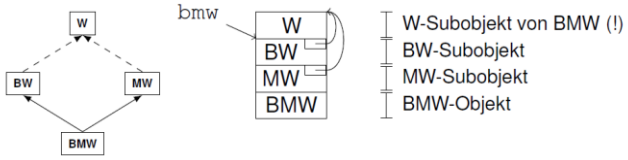
Objekt speichert Subobjekte von allen geerbten Klassen, auch wenn dabei Oberklassen in der Hierarchie doppelt gespeichert werden.



5.2 Virtuelle Vererbung in C++

```
class BW : public virtual W {…}
```

Durch virtual Modifier enthalten Unterklassenobjekte nur einen Zeiger auf das Oberklassenobjekt.

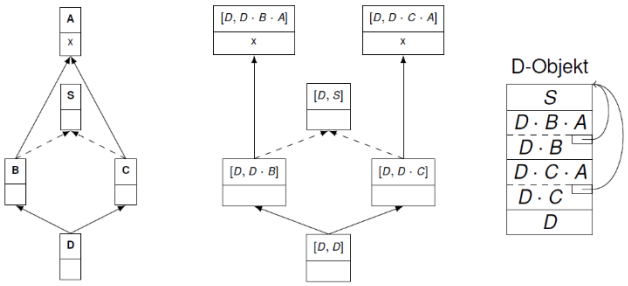


5.3 Subobjektgraphen

Subobjekte sind nur durch vollständige Vererbungspfade eindeutig identifizierbar. Syntax: $[C, C \cdot B \cdot A]$ („Das $C \cdot B \cdot A$-Subobjekt eines C-Objektes“).

Knoten = Subobjekt

Kante (S, S‘) = S‘ ist direkt in S enthalten (nicht-virtuell) oder S enthält einen Zeiger auf S‘ (virtuell). $S \sqsubset_1 S'$.



(S ist nicht mit $D \cdot B \cdot S$ bezeichnet, da durch virtuelle Vererbung nur ein S-Subobjekt in D gespeichert wird.

Bei Einfachvererbung sind Klassengraph und Subobjektgraph gleich.

5.3.1 Konstruktion mittels Induktion

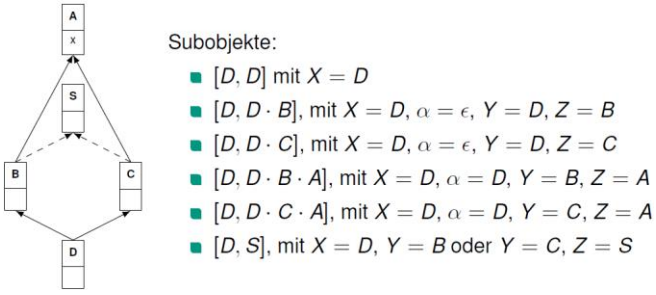
5.3.1.1 Definitionen

Für Klassen X und Y bedeutet:

- $X <_N Y$: X erbt direkt nicht-virtuell von Y
- $X <_V Y$: X erbt direkt virtuell von Y
- $X < Y$: X erbt direkt von Y
- $X <^* Y$: X ist Unterklasse von Y oder X=Y
  - $<^*$ ist reflixiv transitive Hülle von <

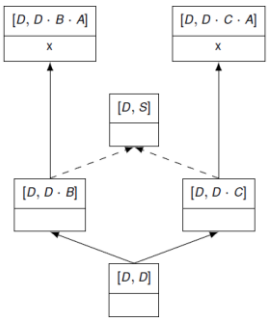
5.3.1.2 Konstruktion der Knoten

- $[X, X]$ ist ein Subobjekt
- Wenn $[X, \alpha \cdot Y]$ Subobjekt und $Y <_N Z$, dann ist auch $[X, \alpha \cdot Y \cdot Z]$ ein Subobjekt.
- Wenn $X <^* Y, Y <_V Z$, dann ist $[X, Z]$ ein Subobjekt.



5.3.1.3 Konstruktion der Kanten

- $[X, \alpha] \sqsubset_1 [X, \alpha \cdot Y]$
- $[X, \alpha \cdot Y] \sqsubset_1 [X, Z]$, wenn $Y <_V Z$



5.4 Static Lookup

$lookup(\sigma, m)$ sucht ab Subobject σ aufwärts das Subobjekt, ab dem m deklariert ist.

5.4.1 SL bei Mehrfachvererbung

Bei mehrdeutigem Erhebnis $lookup(\sigma, x) = \perp$ (also mehrere speziellsten Klassen oberhalb σ deklarieren x)

Formal: $lookup(\sigma, m) = \min(Defs(\sigma, m))$ (s.später)

Komplexes Beispiel der formalen Definition: F.83

TODO F79?

5.4.1.1 Dominanzrelation

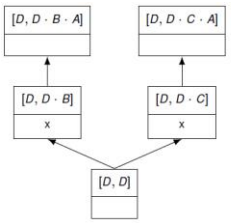
Dominanz: $[C,\alpha] \sqsubseteq [C,\beta]$ wenn es min. einen Pfad vom Subobjektgraphen von α zu dem von β gibt (erlaubt identische Subobjektgraphen, =). $\sqsubseteq$ ist die reflexiv transitive Hülle von $\sqsubset_1$.

Wenn $\forall \sigma' \in S: \sigma \sqsubseteq \sigma'$ (also alle Subobjekte in S über σ erreichbar sind, „Subobjekt σ dominiert Subobjekt-Menge S“) und $\sigma \in S$, dann ist $\sigma = \min(S)$.

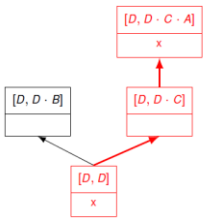
Gibt es kein dominierendes Subobjekt in S, dann $\min(S) = \bot$.

$\min s(S) = \{\sigma \in S | \neg\exists \sigma' \in S: \sigma' \sqsubset \sigma\}$ ist die Menge aller minimalen Elemente von S (min. eines. $mins(S) = \{\sigma\} falls\ \sigma = \min(S) \neq \bot$).

Beispiele:



- Kein Pfad $[D, D \cdot B]$ zu $[D, D \cdot C]$
- Kein Pfad $[D, D \cdot C]$ zu $[D, D \cdot B]$
- Also weder $[D, D \cdot B] \sqsubseteq [D, D \cdot C]$ noch $[D, D \cdot C] \sqsubseteq [D, D \cdot B]$
- Also $min(\{[D, D \cdot B], [D, D \cdot C]\}) = \bot$



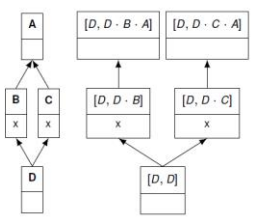
- Pfad $[D, D]$ zu $[D, D \cdot C \cdot A]$
- Also $[D, D] \sqsubseteq [D, D \cdot C \cdot A]$
- Also $min(\{[D, D \cdot C \cdot A], [D, D]\}) = [D, D]$

5.4.1.2 Defs

$$Defs(\sigma, m) = \{\sigma' \sqsupseteq \sigma | m \in Member(ldc(\sigma'))\}$$

ist die Menge aller Subobjekte, die im Subobjektgraph von σ aus erreichbar sind und eine Definition vom Member m enthalten.

- Für „least-derived-class“ Fkt. $ldc([C, \alpha \cdot A]) = A$
- Für Member-Fkt. $Member(C)$ die alle in der Klasse C deklarierten Member zurückgibt.



- $Member(A) = Member(D) = \emptyset$
- $Member(B) = Member(C) = \{x\}$
- $Defs([D, D], x) = \{[D, D \cdot B], [D, D \cdot C]\}$
- $min(Defs([D, D], x)) = \bot$
- $Defs([D, D \cdot B], x) = \{[D, D \cdot B]\}$
- $min(Defs([D, D \cdot B], x)) = [D, D \cdot B]$

5.5 Dynamische Bindung nach Rossie/Friedman

Problem: Vorige Formalisierung betrachtet nur statischen Lookup, dynamische Bindung hängt aber vom dynamischen Typen ab.

$dynBind(\sigma, f) =$ Subobjekt σ', auf dem f aufgerufen wird (oder $\bot$ bei Mehrdeutigkeit).

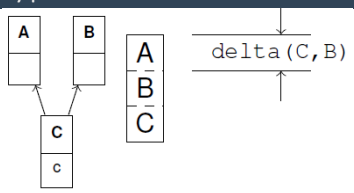
$$dynBind(\sigma, f) = \min\big(Defs([mdc(\sigma), mdc(\sigma)], f)\big)$$
$$= lookup([mdc(\sigma), mdc(\sigma)], f)$$

für „most-derived-class“-Fkt. $mdc([C, \alpha \cdot A]) = C$.

Beispiel: F.88

6 Implementierung von Mehrfachvererbung

6.1 C++ Typecasts



6.1.1 Nicht-virtuelle Einfachvererbung

Nullcode.

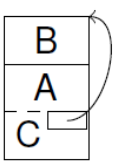
6.1.2 Nicht-virtuelle Mehrfachvererbung

Zeigerverschiebung. Bei $O <_N A, O <_N B$ hat O ein Subobjektlayout von [A,B,O], *Zeiger wird um $delta(B, O) = sizeof\ A$ verschoben auf [B,O], also wird verschoben auf das relevante Subobjekt.*

```
B pb = pc; // pc is type C; wird zu:
B pb = (B*) (((char*)pc) + delta(C,B));
```

$delta(C, B) = sizeof(A)$ ist statisch zur Kompilierzeit bekannt.

6.1.3 Virtuelle Mehrfachvererbung



Verfolgen des Subobjektzeigers. Zeiger von ursprünglichem Objekt wird ersetzt mit Zeiger auf virtuellen Vorgänger.

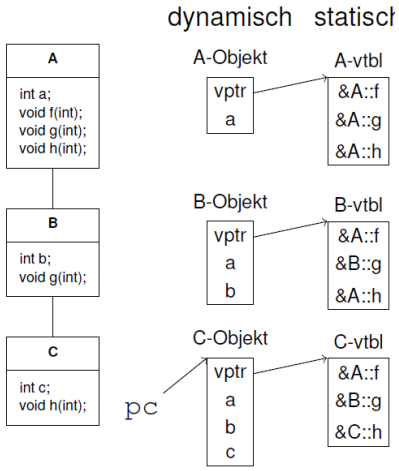
Zur Laufzeit muss geprüft werden dass Zeiger kein Nullptr ist, dieser darf nicht verschoben werden, stattdessen wird Zeiger zum ursprünglichen Objekt selbst mit Nullptr ersetzt.

```
B pb = pc; // pc is type C; wird zu:
B pb = *((B**) (((char*)pc)+offset(B_ptr)));
```

B_ptr ist der Offset des B-Subobjektzeigers im C-Objkt.

6.2 Vererbung mit vtables

6.2.1 Einfachvererbung mit vtables



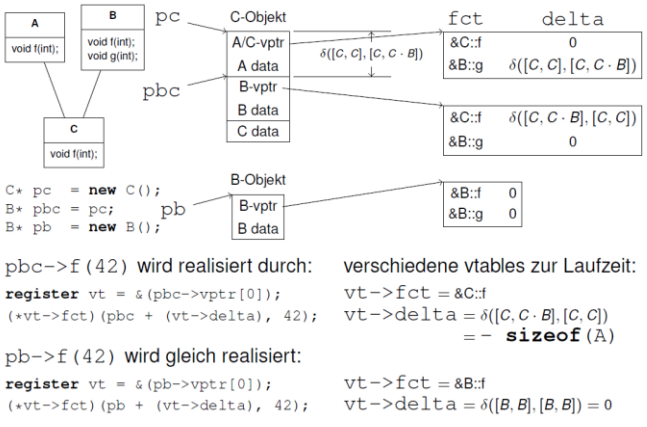
Jede Klasse hat eine statische vtbl mit Einsprungadressen für Methoden. Jedes Objekt enthält Zeiger auf vtbl seiner Klasse.

```
C* c = new C(); c->h(42);
// wird zu:
(*(c->vptr[2]))(c, 42);
```

6.2.2 Mehrfachvererbung mit vtables

```
// C erbt von A und B
// A, B und C definieren alle f()
B* pbc = new C();
pbc->f(42); // ruft C::f(int) auf
```

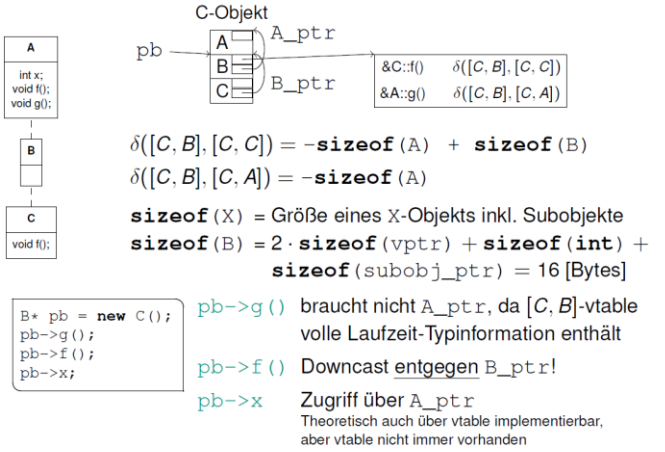
`this`-Zeiger muss auf `C`-Objekt zeigen, pbc zeigt aber auf `B`-Subobjekt eines `C`-Objektes. `this`-Zeiger muss zur Laufzeit gecastet werden mithilfe des Subobjekt-Deltas aus der vtable.



Jedes Subobjekt hat eigenen vptr und vtable.

6.2.2.1 Deltas

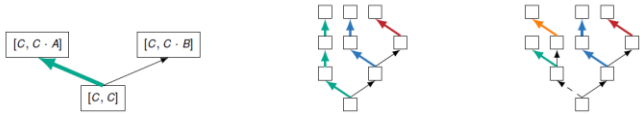
Verschiedene Subobjekte haben verschiedene Offsets, Methodenaufrufe müssen trotzdem funktionieren. ➔ Speichere Subobjektabhängiges δ in vtable, Übersetzer rechnet Deltas beim Kompilieren heraus.



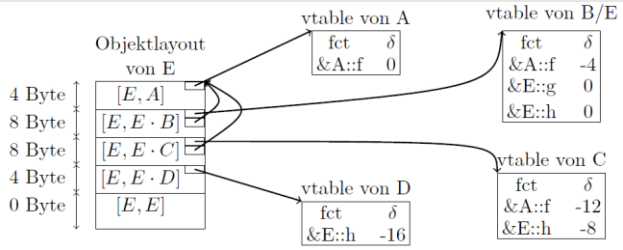
Beispiel: F102ff

6.2.2.2 Gemeinsam benutzte vtables

Vtables können entlang linker Außenkannte im Subobjektgraphen geshared werden: $\delta([x, \alpha \cdot Y \cdot \ldots \cdot Z], [X, \beta]) = \delta([X, \alpha \cdot Y], [X, \beta])$.



6.2.2.3 Beispiel



Beim Delta-Abzählen: Verwende obere Kante eines Subobjekten als Zählreferenz. Achtung: Members von betrachteter Klasse werden meist nicht in dessen Subobjekt, sondern einer geteilten vtable gespeichert.

TODO F147-156

6.3 Umsetzung von CPP Aufrufen

```
pbc->f(42);
```

wird umgesetzt durch:

```
register vt = &(pbc->vptr[K]);
(*vt->fct)(pbc + (vt->delta), 42);
```

Mit K als Index der Methode in der VTable. Parameter kann auch weggelassen werden.

7 Java Interfaces

7.1 Implementieren von Interfaces

7.1.1 Ansatz: C++-Strategie

Wenn zur Compilezeit komplette Interface-Hierarchie bekannt ist: Betrachte Interfaces als zusätzliche Oberklassen mit eigenem vptr pro implementiertem Interface (Subobjekt besteht nur aus vptr da keine Instanzvariablen im Interface). Eigene vtable pro Klassen-Interface-Paar.

Contra: Hoher Overhead pro Objekt (viele implementierte Interfaces ➔ viele vtables), weiterhin this-Zeiger-Verschiebung notwendig.

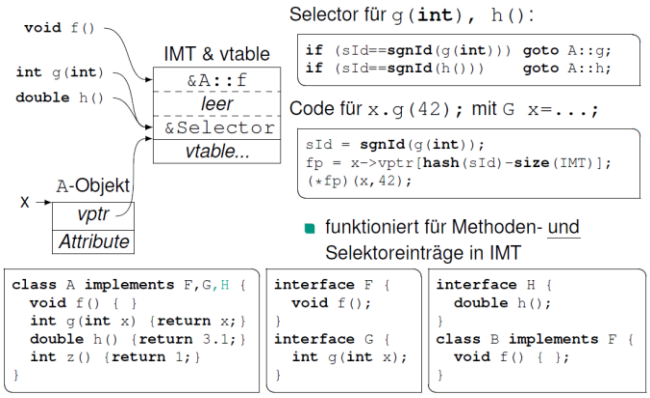
7.1.2 Problem: Dynamisches Laden von Klassen/Interfaces

In Java ist Klassen-/Interfacehierarchie nicht zur Compilezeit bekannt. Für Interfacemethoden sind daher keine klassenhierarchieweit eindeutigen Methodenindizes möglich.

7.1.3 Interface Method Tables

Tabelle die dynamisches Nachladen von Interfacehierarchien erlaubt. Werden bei erster Verwendung der Klasse aufgebaut.

Methodensignaturen (zB `f(int)`) werden auf IDs zugeordnet. Methoden (zB `f`) werden auf Hashs zugeordnet. Kollisions-Blöcke werden mit `Selector`-Code beschrieben (Programmcode der für Kollisionsmöglichkeiten separat entscheidet).



IMT ist eine Hashtabelle fester Größe vor der vtable, IMT-Anfangsadresse berechnet sich aus vptr und size(IMT).

7.2 Java: Default-Methoden

Siehe Folien F112.

8 Überladungen

```
static void print(int i) { ... }
static void print(double d) { ... }
static void print(String s) { ... }
```

Jede Variante hat eigenen vtable-Eintrag.

- C++: Erst Static Lookup ohne Methodensignatur, dann Überladungsauflösung innerhalb gefundener Klasse
- Java: Static Lookup mit Methodensignatur. Erst Auswahl der spezifischsten Signatur unter sichtbaren Methoden, zur Laufzeit dynamische Bindung für Methode inkl. Signatur.

```
class O {
  void show(int i) {}
  void show(double d) {} }
class U extends O {
  void show(double d) {} }

O o = new O();
U u = new U();

o.show(1.0); // O::show( double )
o.show(17); // O::show( int )
u.show(2.0); // U::show( double )
u.show(42); // Unterschiedlich !
((O)u).show(23); // O::show( int )
((O)u).show(3.0); // U::show( double )
```

Dynamische Bindung greift bei `((O)u).show(3.0)`.

8.1 Spezifischste Methode

Bei mehreren Kandidaten wählt der Compiler die spezifischste Methode (*Methode bei denen die Parameter am tiefsten in der Erbungshierarchie sind*). Bei Mehrdeutigkeit kann nicht kompiliert werden.

8.2 Überladung und dynamische Bindung

Dynamische Bindung respektiert Überladung, und erzwingt nicht das Nutzen einer Methode einer erbenden Klasse mit einer anderen Signatur. (Siehe F.122)

8.3 Smart Pointer

Voll dynamische Typisierung: Man möchte ein Objekt, das zur Laufzeit seinen Typ wechseln kann.

Realisierung durch Role-Pattern (Unterklassen erben von Rolle, Objekt enthält wechselbare Referenz auf Rolle) oder Smart Pointer: Objekt enthält Referenz auf wechselbare Rolleninstanz, Dereferenzierungs-

operator des Objektes ist überladen und zeigt auf die Rolleninstanz. (F.127)

9 Innere Klassen

9.1 Statische innere Klasse

Innere und äußere Klasse können jeweils auf statische Member zugreifen. Innere Klasse kann auch auf private statische! Member der äußeren zugreifen. Äußere Klasse muss nicht statisch sein.

Nicht-statische innere Klassen dürfen selbst keine statischen inneren Klassen/sonstige Deklarationen enthalten. (Äußere nicht statische Klassen schon)

9.2 Dynamische innere Klasse

Innere Klasse kann auf (auch private) nicht-statische und statische Member der äußeren Klasse zugreifen. Innere Klasse hat implizite Referenz auf Instanz der äußeren Klasse mit `OuterClass.this`.

> Dynamische innere Klassen sind statisch gebunden. (TODO verify) Virtuelle innere Klassen dagegen sind dynamisch gebunden.

9.3 Anonyme Klassen

Anonyme und lokale Klassen erben das static-Attribut von der umgebenden Methode und können in diesem Fall nicht auf nicht-statische Member der Oberklasse zugreifen.

Innerhalb von anonymen und lokalen Klassen darf wie gewohnt auf Members der umschließenden Klasse zugegriffen werden, auf lokale Variablen und Parameter der umschließenden Methode kann jedoch nur dann zugegriffen werden, wenn diese als `final` deklariert sind oder effektiv final sind.

9.4 Beispiel

Siehe Folien F.134 zu zyklisch einfach verketteten Listen.

10 Generics

Bounded Polymorphism (Java, nicht C++): Parametertyp muss Unterklasse einer bestimmten Klasse sein. So können Methoden der generischen Klasse typgecheckt werden. („Typschranke“).

Typschranken dürfen beim Erben spezieller werden (aber nicht allgemeiner oder unvergleichbar).

```
class A<T> {}
class B<T extends Other>
  extends A<T> {}
```

Mehrere Typschranken möglich.

```
class A<T extends X & Y> {}
```

Wildcards: `?` ist anonymer Typparameter, mehrere `?` werden als verschiedene Parameter aufgenommen. Obere Schranken (`(? extends C)`, Lesen mit Typ C möglich) und *untere Schranken* (`(? super D)`, Zuweisen mit Typ D möglich) können angegeben werden. (Beispiele siehe F.149)

```
static <T> int binarySearch(
  List<? extends Comparable<? super T>>
    list, T elem)
```

Type Erasure: Weil Generics erst ab Java 1.4 enthalten sind, werden wegen Kompatibilität Typen bei Compilierung nach Typprüfung entfernt und zu Downcasts zu `Object` ersetzt.

10.1 PECS

Remember *PECS*: **"Producer Extends, Consumer Super":**

- **Producer Extends** - If you need a `List` to produce `T` values (you want to read `T`s from the list), you need to declare it with `? extends T`, e.g. `List<? extends Integer>`. But you cannot add to this list.
- **Consumer Super** - If you need a `List` to consume `T` values (you want to write `T`s into the list), you need to declare it with `? super T`, e.g. `List<? super Integer>`. But there are no guarantees what type of object you may read from this list.

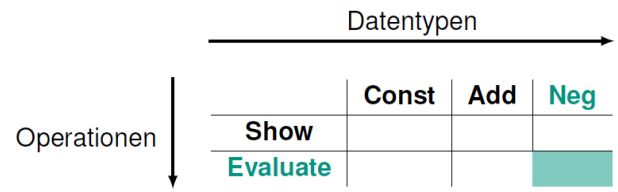
If you need to both read from and write to a list, you need to declare it exactly with no wildcards, e.g. `List<Integer>`.

10.1.1 Example

Note how the source list `src` (the producing list) uses `extends`, and the destination list `dest` (the consuming list) uses `super`:

```
public class Collections {
 public static <T> void copy(List<? super T> dest,
                             List<? extends T> src) {
   for (int i = 0; i < src.size(); i++)
     dest.set(i, src.get(i));
 }
}
```

11 Tyrannei der dominanten Dekomposition



Idee: Neue Operationen und Datentypen können sicher und einfach eingeführt werden.

- **Einfach**: *Lokalitätsprinzip*, keine Veränderungen des bestehenden Codes, Erweiterung in isolierten Dateien
- **(Typ-)Sicher**: Keine Downcasts oder instanceof, Anwendung von nicht angepasster Operation auf neuen Datentyp muss statisch abgelehnt werden.

11.1 Ansätze

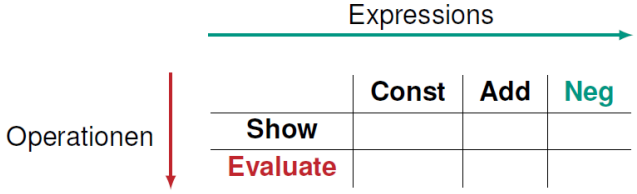
11.1.1 Datentypen als Klassenhierarchie, Operationen als dynamisch gebundene Methoden

```
interface Expr { String show(); }
class Const implements Expr {...}
class Add implements Expr {...}
```

Neue Datentypen können als von `Expr` erbende Klassen *einfach* und *typsicher* eingeführt werden. **Neue Methoden** können im Interface `Expr` deklariert und so *typsicher* eingeführt werden, aber *nicht einfach* (alle Klassen müssen geändert werden).

11.1.1.1 Erbendes Interface

Statt `Expr` direkt eine neue Methode dazuzufügen, neues von `Expr` erbendes Interface `EvalExpr` definieren mit neuer Methode. **Neue Methoden** können dann *einfach*, aber *nicht typsicher* eingeführt werden (siehe Beispiel F.161, in neuer Methode können keine `EvalExpr` Member benutzt werden, nur `Expr`. Downcast nötig!).



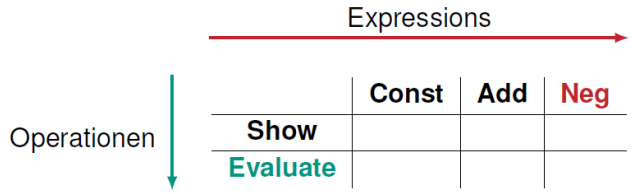
11.1.2 Visitor-Pattern

```
interface Expr { <T> T accept(Visitor<T> v); }
class Const implements Expr {
  int value;
  Const(int v) { value = v; }
  public <T> T accept(Visitor<T> v) { return v.visit(this); } }

class Add implements Expr {
  Expr left, right;
  Add(Expr l, Expr r) { left = l; right = r; }
  public <T> T accept(Visitor<T> v) { return v.visit(this); } }

interface Visitor<T> {
  <T> T visit(Const c);
  <T> T visit(Add a); }

class PrettyPrinter implements Visitor<String> {
  public String visit(Const c) {return String.valOf(c.value);}
  public String visit(Add a) {
  return "(" + a.left.accept(this) + " + "
    + a.right.accept(this) + ")"; } }
```

Neue Operationen können als erbende Klassen von `Visitor<Integer>` *einfach* und *typsicher* realisiert werden.

Neue Datentypen können durch neue erbende Klasse von `Expr` *typsicher*, aber *nicht einfach* realisiert werden (neue `visit` Methode im `Visitor` Interface und allen davon erbenden Klassen notwendig).

Wieder Reparaturversuch durch von Visitor erbender NegVisitor, aber selbes Problem wie davor, einfach aber nicht typsicher da notwendige Downcasts.



11.2 Problem

Eine Hierarchie dominiert die andere, die dominante Hierarchie ist einfach, die dominierte ist schwer zu ändern.

- Objektorientierte Dekomposition: Operationen als dynamisch gebundene Methoden, neue Datentypen einfach, aber neue Operationen nicht.
  ⇨ Datentypen dominieren Operationen
- Funktionale Dekomposition: Operationen als Visitor-Objekte, neue Operationen einfach, neue Datentypen nicht.
  ⇨ Operationen dominieren Datentypen.

11.3 Lösungsansätze

11.3.1 Multimethoden

Existiert nicht in Java, aber zB in Variante MultiJava.

```
abstract class Shape {
  boolean intersects(Shape other) {throw …} }
class Rect extends Shape {
  boolean intersects(Shape@Rect other) {...}
  boolean intersects(Shape@Circle other) {...} }
class Circle extends Shape {
  boolean intersects(Shape@Rect other) {...}
  boolean intersects(Shape@Circle other) {...} }
```

Dynamische Bindung bestimmt anhand Bezugsobjekt das Subobjekt, aus dem Methodendefinition kommt.

Compiler erzeugt dann dynamisch Fallunterscheidung (`if s instanceof Rect…`), dh. es entsteht keine Überladung, sondern aus allen Multimethoden wird eine Methode erzeugt.

- Neue Datentypen einfach
  - `class Neg impl. Expr { ... }`
  - `class NegPrettyPrint extends PPrint {`
  - `  String apply(Expr@Neg n) {...} }`
- Neue Operation einfach
  - `class Evaluator {`
  - `  // .. old methods`
  - `  int apply(Expr@Add a) {...} }`

Einfaches einführen neuer Konzepte, aber immernoch nicht typsicher (Typ wird zur Laufzeit bestimmt).

11.3.2 Traits, Mixins, abstrakte Typmember

Optional, Siehe Unterkapitel zu „Scala“.

11.3.3 Virtuelle Klassen

Ähnlich wie innere Klassen, aber Redefinitionen und dynamische Bindung bei Vererbung möglich.

```
class A {
  class C { public void print() { print("A::C"); } } }
class B extends A {
  class C { public void print() { print("B::C"); } } }
A a = new B();
a.C c = a.new C(); // C ist dynamisch gebunden !
c.print(); // Ausgabe : "B::C"!
```

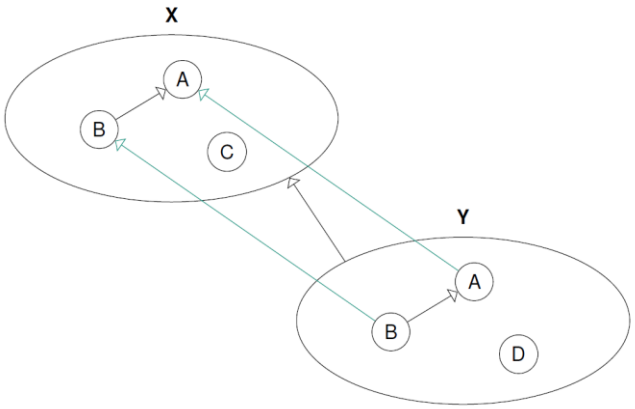
Virtuelle Klasse kann mit `out` auf die umgebende Klasseninstanz verweisen, zB. `out.prop`.

```
class A {
  class C { public void print() { print("A::C"); } }
  class D { out.C c = out.new C(); } }
class B extends A {
  class C { public void print() { print("B::C"); } } }
 A a = new B();
 a.D d = a.new D();
 d.c.print(); // Ausgabe: "B::C"
```

11.3.3.1 Familienvererbung

```
class X {
  class A { int a = 42; }
  class B extends A { int b = 23; } }

class Y extends X {
  class A {// erbt von X::A
    int getA() { return a; } }
  class B extends A {// erbt von Y::A, X::B
    int getB() { return b; } } }
```

> Wenn eine Oberklasse von einer anderen erbt, erben alle virtuellen Klassen implizit von den redefinierten virtuellen Klassen der der vererbten Oberklasse.



TODO FG.270ff

> Virtuelle Klassen sind im Gegensatz zu inneren Klassen dynamisch gebunden. (TODO verify)

11.3.3.2 Effiziente Implementierung

Virtuelle Klassen lassen sich implementieren indem dynamische Bindung der Konstruktoren der inneren Klassen erlaubt wird.

11.3.4 Objektschnittstellen

TODO

Siehe Übungsblatt 10 A3

12 Programmanalyse

Ziel: Optimierung (Eliminierung von dynamischer Bindung, statischer Methodenaufruf ist effizienter, tote Methoden entfernen, …) und Fehlersuche.

12.1 Eigenschaften

- Statische Analyse: Kompilat wird nicht ausgeführt, nur analysiert

- Whole-Program-Analyse: Gesamtes Programm und Klassenhierarchie wird vorausgesetzt, notwendig für zB Entfernen toter Methoden.

Konservative Approximation: Falls eine Eingabe existiert, bei der das Programm ein bestimmtes Ablaufverhalten zeigt, muss die Analyse dieses Verhalten berücksichtigen.

- Überapproximation: Analyse darf zu viele Aufrufziele analysieren
- Unterapproximation: Menge der entfernbaren Klassen darf zu klein sein

Korrektheitsbedingung bei Kontrollflussgraphen (CFG): Für alle Eingaben z durchläuft das Programm dynamisch die Folge von Grundblöcken (Formal: $W = (Start, b_1, \dots, b_n, End) \Rightarrow$ W ist Pfad im CFG). Damit sind CFGs konservative Approximationen.

12.1.1 Fluss-Sensitivität

Fluss-sensitiv: Beachtet Reihenfolge von Befehlen und berechnet Analyseergebnis pro Programmzeile. Präziser und aufwendiger.

Fluss-insensitiv: Ignoriert Befehlsanweisungen und berechnet Analyseergebnis pro Programm. Unpräziser, aber effizienter.

12.1.2 Kontext-Sensitivität

Kontext-sensitiv: Beachtet Aufrufkontext einer Methode, berechnet Analyseergebnis pro Aufrufkontext und pro Methode. („Bei Aufruf von f kann Zeiger x nur auf Instanz o1 zeigen“). Präziser und aufwendiger.

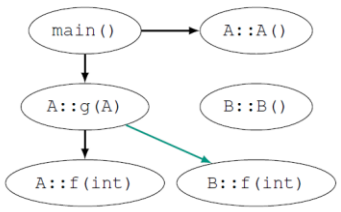
Kontext-insensitiv: Ignorieren Aufrufkontext, berechnen Analyseergebnis pro Methode. Unpräziser, aber effizienter.

12.2 Rapid Type Analysis

12.2.1 Call-Graph

Analysiere Aufrufstruktur als Graphen. Knoten sind Methodensignaturen, Kanten geben an ob innerhalb einer Methodensignatur eine andere aufgerufen werden *kann*. Wegen dynamischer Bindung müssen alle potentielle Ziele berücksichtigt werden.

```
1  class A {
2    void f(int x) { }
3    void g(A p) {p.f(42);}
4  }
5  class B extends A {
6    void f(int x) { }
7  }
8  void main() {
9    A p = new A();
10   p.g(p);
11 }
```



Umgang mit dynamischer Bindung: Wegen konservativer Approximation, Bestimme Menge Z der potentiellen Aufrufziele: finde aufgerufene Methode durch Static Lookup (Aufwärtssuche), füge aufgerufene Funktion und *alle* Methoden mit passender Signatur abwärts der Hierarchie dazu.

```
class A { void f() {} void g() {} }
class B extends A { void f() {} }
class C extends B { void g() {} }
class D extends B { void f() {} void g() {} }
B b = ...;
b.g(); // Z = { A::g(), C::g(), D::g() }
b.f(); // Z = { B::f(), D::f() }
```

12.2.2 Reduktion der Call-Graph-Größe (RTA)

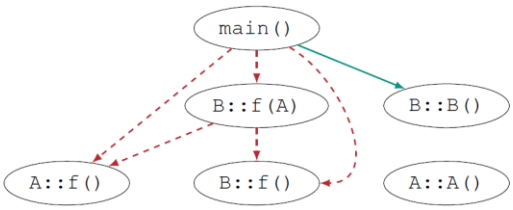
Durch dynamische Bindung sind viele Kanten unnötig (zB `A::g(A)` → `B::f(int)`). Solche können mit Informationen über nie aufgerufene Konstruktoren entfernt werden.

1. Markiere alle Kanten zu virtuellen Methoden als verboten, alle Kanten von `main()` zu Konstruktoren als erlaubt.
2. Wiederhole, bis keine erlaubten Kanten vorkommen:
   a. Traversiere Graphen von `main()` entlang erlaubter Kanten und markiere erreichbare Knoten.
   b. Bei erreichen eines Konstruktors `A::A()`, markiere Kanten von markierten Knoten zu Methoden in `A`'s *VTable* als erlaubt.
3. Behalte nur von `main()` über erlaubte Kanten erreichbare Methoden.

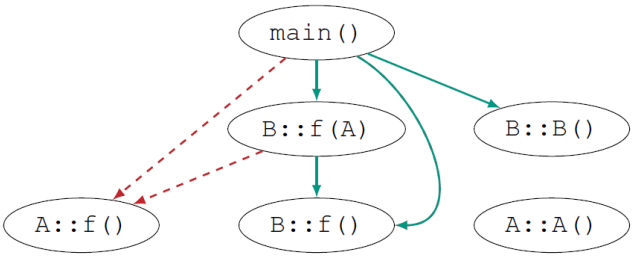
Achtung: VTable von A kann auch aus Oberklassen ererbte Methoden enthalten, was zu dennoch redundanten Kanten führt.

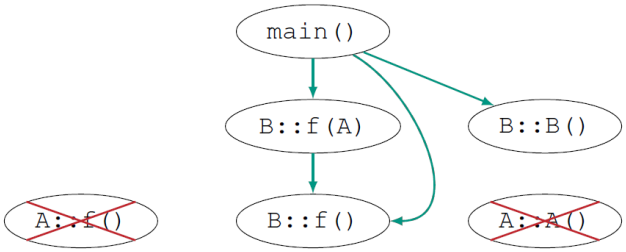
```
class A {
  int f() { return 42; }
}
class B extends A {
  int f() { return 17; }
  int f(A x) { return x.f(); }
}
```

```
void main() {
  B p = new B();
  int r1 = p.f(p);
  int r2 = p.f();
  A q = p;
  int r3 = q.f();
}
```



erlaubte Kanten verbotene Kanten





12.2.3 RTA als Constraint Problem

Finde Mengen $R = LebendigeMethoden$ und $S = LebendigeKlassen$ mit:

$$\frac{}{main \in R} \qquad \frac{}{Main \in S} \qquad \frac{M \in R \quad new\ C() \in body(M)}{C \in S}$$

$$\frac{M \in R \wedge e.m(x) \in body(M) \wedge C \leq staticType(e) \wedge C \in S \wedge lookup(C, m) = M'}{M' \in R}$$

12.2.4 Fazit

- Sehr schnell, da *fluss- und kontext-insensitiv*.
- Effektiv bei Klassenbibliotheken (viel kann entfernt werden)
- Sehr ungenau: Da *keine Zeiger-Analyse* können oft dynamisch gebundene Aufrufe nicht aufgelöst werden.

12.3 Points-To-Analysis

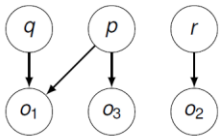
Bestimme für jeden Zeiger, auf welche Objekte er zeigen kann.

12.3.1 Points-To-Graph

Knoten: Entweder Objektrepresäntant oder Zeigervariable.

Kante: $p \rightarrow o_i$ wenn p bei irgendeiner Ausführung auf o_i zeigen könnte.

```
void main() {
  Object q = new Object();   // o1
  for (int i = 0; i < 10; ++i)
    Object r = new Object(); // o2
  Object p = q;
  if (Math.random() < 0.5)
    p = new Object();        // o3
}
```



12.3.2 Zuweisung nach Andersen

Fluss-sensitiv und Kontext-sensitiv.

Wie geht man mit Zuweisungen (`p = q;`) um?

> Zuweisung `p=q;` induziert $PT(q) \subseteq PT(p)$. (p kann auf alles zeigen, auf das q zeigen kann). Gesucht sind Mengen $PT(p), PT(q)$, die alle derart konstruierten Ungleichungen erfüllen.

Neue Objekte (`new Object()`) werden mit Repräsentation $\{o_i\}$ instantiiert.

```
void main() {
  Object p = new Object(); // {o1} ⊆ PT(p)
  Object q = p;            // PT(p) ⊆ PT(q)
  Object r = q;            // PT(q) ⊆ PT(r )
  p = r;                   // PT(r) ⊆ PT(p)
  Object s = new Object(); // {o2} ⊆ PT(s)
  r = s;                   // PT(s) ⊆ PT(r )
}
// Lösung: PT(P) = PT(q) = PT(r) =
// {o1, o2}, PT(s) = {o2}
```

12.3.3 Lösung des Mengengleichungssystems

Für die daraus ergebenden Mengenungleichungssysteme $M_i \subseteq N_j$ (i=j möglich), sind die kleinsten Mengen M_i, N_j gesucht, die alle Ungleichungen erfüllen.

> Dazu werden Mengen M_i, N_j als Knoten und Ungleichungen $M_i \subseteq N_j$ als Kanten $M_i \rightarrow N_j$ modelliert.

12.3.3.1 Trivialer Ansatz

1. Initialisiere alle unbekannten $M_i, n_j = \emptyset$
2. Traversiere Graph, für jeden Knoten X:
   a. Für bereits berechnete Vorgänger $Y_1, \dots, Y_i$ berechne $X := X \cup (\bigcup_i Y_i)$
3. Falls sich min. ein Knoten geändert hat →#2.

Ineffizient, $O(k^4)$ bei k Ungleichungen, Zyklen brauchen lange bis sie stabil sind.

> Beispiel: F227

12.3.4 Zugriffe auf Attribute

Modellierung von Objektattributen.

Lesen `x = p.f;`	$\frac{o_i \in PT(p)}{PT(o_i.f) \subseteq PT(x)}$
Schreiben `p.f = y;`	$\frac{o_i \in PT(p)}{PT(y) \subseteq PT(o_i.f)}$

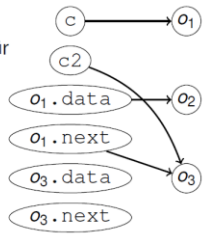
Also: Wenn x von p.f liest und p auf o_i zeigen kann, füge $o_i.f$ zur PT-Menge von x dazu. Wenn p.f von y liest und p auf o_i zeigen kann, füge y zur PT-Menge von $o_i.f$ dazu.

```
1 class C { Object data; C next; }
2 C c = new C();          // o1 mit PT-Mengen für o1.data, o1.next
3 c.data = new Object(); // o2
4 c.next = new C();       // o3 mit PT-Mengen für o3.data, o3.next
5 C c2 = c.next;
```

Regeln für Attributzugriffe (bedingte Mengenungleichungen, müssen für jedes (typkorrekte) o_i aufgestellt werden!):

Lesen `x = p.f;` $\frac{o_i \in PT(p)}{PT(o_i.f) \subseteq PT(x)}$

Schreiben `p.f = y;` $\frac{o_i \in PT(p)}{PT(y) \subseteq PT(o_i.f)}$



12.3.4.1 Systeme mit bedingten Ungleichungen

Bedingte Ungleichung: Wie oben aufgestellt, Ungleichung gilt wenn $o_i \in PT(p)$.

1. Initialisiere Knoten (ein Knoten pro Menge) und Kanten ($M_i \to N_j$ für Ungleichung $M_i \subseteq N_j$). Markiere Kanten für unbedingte Ungleichungen als erlaubt, Kanten für bedingte Ungleichungen als verboten.
2. Propagation: Traversiere Graph in beliebiger Reihenfolge, berechne für jeden Knoten X und seine Vorgänger Y_i das Ergebnis $X \coloneqq X \cup (\bigcup_i Y_i)$
3. Bedingte Ungleichungen: Falls X sich geändert hat, prüfe alle Ungleichungen mit X in der Bedingung und markiere evtl neue Kanten als erlaubt. (Während des Lösens werden neue Kanten freigeschaltet!)
4. Terminiere, wenn sich nichts mehr ändert.

Beispiel: F230

Komplexität von $O(n^3)$ bei n Mengen, in der Praxis unter $O(n^2)$.

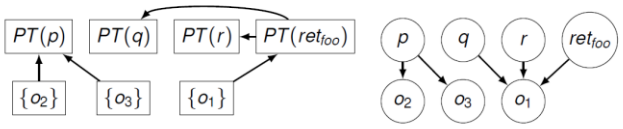
12.3.5 Umgang mit statischen Methodenaufrufen

Behandle jedes $return\ e$ innerhalb einer Methode m als Zuweisung $ret_m = e$ an Hilfsvariable ret_m.

Behandle jeden Methodenaufruf $q = m(a)$ als Zuweisung der Argumente $p = a$ und Zuweisung des Rückgabewerts $q = ret_m$.

Im Beispiel: kontext-insensitive Points-To-Analyse nach Andersen berechnet eine PT-Menge pro Parameter für alle Aufrufkontexte.

```
1 class C {
2   static Object foo(Object p) {
3     return new Object();          // o1   {o1} ⊆ PT(ret_foo)
4   }
5 }
6 Object q = C.foo(new Object()); // o2   {o2} ⊆ PT(p), PT(ret_foo) ⊆ PT(q)
7 Object r = C.foo(new Object()); // o3   {o3} ⊆ PT(p), PT(ret_foo) ⊆ PT(r)
```



Kontext-sensitive Analyse könnte hier zwischen Aufrufkontexten unterscheiden und würde $PT(p) = \{o_2\}$ für ersten Aufruf in Zeile 6 und $PT(p) = \{o_3\}$ für zweiten Aufruf in Zeile 7 liefern.

12.3.6 Umgang mit dynamischen Methodenaufr.

Methodenaufruf `q = o.m(a)` bildet bedingte Menge:

$$\frac{o_i \in PT(o) \quad type(o_i) = C \quad lookup(C,m) = R\ \ D :: m(T\ \ p)}{PT(a) \subseteq PT(p) \quad \{o_i\} \subseteq PT(this_m^D) \quad PT(ret_m^D) \subseteq PT(q)}$$

Der lookup wird implizit durchgeführt und bei der Bedingung nicht mit dazugeschrieben, sondern verwendet um den Typ D für die Folgerung herauszufinden. Für p wird die (potentiell zuvor initialisierte) Parametervariable aus dem resolvierten Methodenaufruf verwendet. o_i wird nicht explizit aufgelöst. Es wird *für jede mögliche Klasse C eine Gleichung* aufgestellt. Der Lookup geschieht statisch, aber typkorrekt zum aktuell betrachteten Typ von o_i!

Achtung: ret_m^D und $this_m^D$ nicht verwechseln!

Beispiel: F236

12.3.7 PT nach Andersen Übersicht

Zuweisung neues Objekt `O x = new O();`	$\{o_1\} \subseteq PT(x)$
Zuweisung Variable `p = q;`	$PT(q) \subseteq PT(p)$

Return einer Funktion
`class O {`
`  R func() {return x;}}`
	$PT(x) \subseteq PT(ret_f^O)$

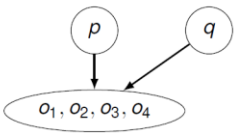
Aufruf dynamische Funktion `q=o.m(a)` $\forall C: Class$ $\frac{o_i \in PT(o) \quad type(o_i) = C \quad lookup(C,m) = R\ \ D :: m(T\ \ p)}{PT(a) \subseteq PT(p) \quad \{o_i\} \subseteq PT(this_m^D) \quad PT(ret_m^D) \subseteq PT(q)}$ `mit statischem Lookup von typkorrekten o_i!`	
Attribut Lesen `x = p.f;`	$\frac{o_i \in PT(p)}{PT(o_i.f) \subseteq PT(x)}$
Attribut Schreiben `p.f = y;`	$\frac{o_i \in PT(p)}{PT(y) \subseteq PT(o_i.f)}$

12.3.8 Zuweisung nach Steensgaard

Ziel: Ungenauerer aber kleinerer Graph.

Bei Zuweisung $p = q$ tue so, als ob auch umgekehrte Zuweisung $q = p$ vorhanden ist. Damit $PT(q) \subseteq PT(p) \wedge PT(p) \subseteq PT(q) \Leftrightarrow PT(p) = PT(q)$, also werden *PT(p) und PT(q) verschmolzen und bilden eine Äquivalenzklasse*.

```
1 void main() {
2   Object p = new Object(); // o1
3   if (Math.random() < 0.5)
4     p = new Object();      // o2
5   else
6     p = new Object();      // o3
7   Object q = new Object(); // o4
8   p = q;
9 }
```

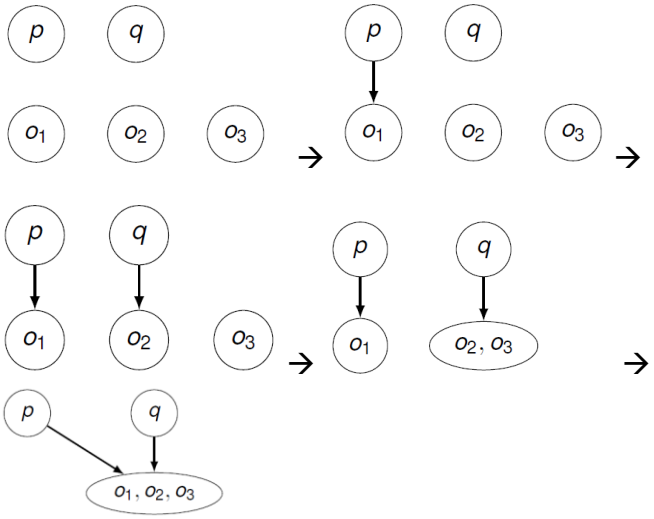


Damit ist der Graph deutlich simpler und ein effizienterer Algorithmus ist möglich.

12.3.8.1 Effizienter Steensgaard-Algorithmus

Verwende Union-Find-Datenstruktur. Am Anfang sind alle Objektrepräsentationen o_i in eigenen Partitionen. Bei Zuweisung $p = q$ werden Partitionen $PT(p)$ und $PT(q)$ vereinigt.

```
1 void main() {
2   Object p = new Object(); // o1
3   Object q = new Object(); // o2
4   q = new Object();        // o3
5   q = p;
6 }
```



13 Typsysteme für Objektorientierung

13.1 Grundlagen

13.1.1 Regelsysteme

Siehe Folien, auch Regeln zu Speicherzellen auf F248 und Typregeln für Records auf F250.

13.1.2 Lambda-Kalkül

Siehe Folien

13.1.2.1 Speicherzellen

Erweiterung des Lambda-Kalküls um Zustand und Seiteneffekte zu modellieren

- Allokation: $new\ t$, zB $new\ 42, new\ true$
- Lesen: $!\,t$, zB $\lambda r.!\,r, !\,(new\ 17) + 5$
- Schreiben: $t_1 := t_2$, zB $new\ 0 := 23$
- Typ Speicherzelle: $Cell\ \tau$
- Typ Dummy: $unit$

13.2 Objekte als einfache Records

Objektinstanzen werden als Records von Typen modelliert, Attribute zu Speicherzellen, final-Attribute zu normalen Membern, Methoden zu Funktionen.

Beispiel:

```
class C {
  boolean b; final int a;
  int f(int x) {return x+3;};
  C(boolean b1, int a1) {} }
```

$$\tau_C = \{b: Cell\ bool, a: int, f: int \to int\}$$

$$C = \lambda b_1.\lambda a_1.\{b = new\ b1, a = a1, f = \lambda x.x + 3\}$$

Eine erbende Klasse enthält auch die Members der Oberklasse in der Modellierung.

13.3 Typkonversionen

Typkonversion $\sigma \leq \tau$: Jedes σ-Objekt ist auch ein τ-Objekt (Unterklassenbeziehung). $\leq$ ist eine Halbordnung, also reflexiv ($\sigma \leq \sigma$), transitiv ($\sigma \leq \sigma' \wedge \sigma' \leq \sigma'' \Rightarrow \sigma \leq \sigma''$) und antisymmetrisch ($\sigma \leq \sigma' \wedge \sigma' \leq \sigma \Rightarrow \sigma = \sigma'$).

Achtung: Polymorphe und monomorphe Typen sind nicht vergleichbar!

13.3.1 Typkonversionen für Objekte

Objektkonversion:

$$O1\ \frac{\sigma_1 \leq \tau_1 \quad \dots \quad \sigma_n \leq \tau_n}{\{m_1: \sigma_1, \dots, m_n: \sigma_n\} \leq \{m_1: \tau_1, \dots, m_n: \tau_n\}}$$

Objekterweiterung:

$$O2\ \frac{}{\{m_1: \tau_1, \dots, m_n: \tau_n, m_{n+1}: \tau_{n+1}, \dots, m_{n+k}: \tau_{n+k}\} \leq \{m_1: \tau_1, \dots, m_n: \tau_n\}}$$

Beispiel: $\{a: int, b: int, c: bool\} \leq \{a: int, b: double\}$

Also $O_1 \leq O_2$ wenn alle Member von O_1 die $\leq$-Relation für alle Member von O_2 erfüllen, oder $O_1 \leq O_2$ wenn O_1 identisch zu O_2 aber mehr Members enthält.

TODO Wann gelten Subtypbeziehungen nur zwischen Instanzen, nicht zwischen Objekten? Siehe Blatt 13 A3 d.

Subsumption/Typkonformanz: TODO Beschreibung

$$SUB\ \frac{\Gamma \vdash t:\sigma \quad \sigma \leq \tau}{\Gamma \vdash t:\tau}$$

13.3.2 Typkonversionen für Methoden

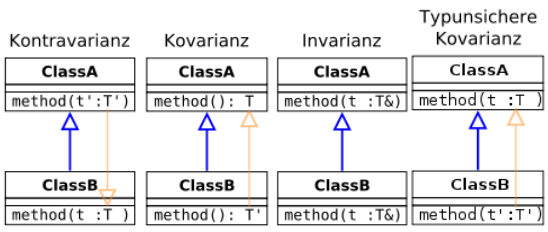
Parameter dürfen nur allgemeiner, Rückgabetypen nur spezieller werden.

Bespiel: F255

TODO Formel F255/FG382

13.4 Ko- und Kontravarianz bei Vererbung

TODO?



13.5 Typkonstruktor

Komplexer Typ der aus gegebenen Typen einen neuen Typ konstruiert (Typen von Klassenkonstruktoren).

$$\_[] : T \mapsto T, \tau \mapsto \tau[]$$

$$Cell\_ : T \mapsto T, \tau \mapsto Cell\ \tau$$

13.6 Array-Anomalie in Java

Arraykonstruktor _[] ist in Java kovariant, kann aber zu Problemen führen durch zB

```
Ober[] o = new …; Unter[] u = o; u[0] = new Ober();!
```

Subtypen bei Typkonstruktoren, ~~Cardelli-Typsystem~~:

- Kovarianz verbietet Schreibzugriffe. (TODO Formeln)
- Kontravarianz verbietet Lesezugriffe. (TODO Formel)

TODO Weitere Typkonstruktoren

13.7 Regeln Referenz

Typsystem $\Gamma \vdash t:\tau$ bedeutet: Im Typkontext Γ hat Term t den Typ τ.

$CONST\ \frac{c \in Const}{\Gamma \vdash c:\tau_c}$	
$VAR\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x:\tau}$	
$ABS\ \frac{\Gamma, x:\sigma \vdash t:\tau}{\Gamma \vdash \lambda x.t:\sigma \to \tau}$	
$APP\ \frac{\Gamma \vdash t_1:\sigma \to \tau \quad \Gamma \vdash t_2:\sigma}{\Gamma \vdash t_1\ t_2:\tau}$	
$NEW\ \frac{\Gamma \vdash t:\tau}{\Gamma \vdash new\ t:Cell\ \tau}$	Allokation einer Speicherzelle.
$!\ \frac{\Gamma \vdash t:Cell\ \tau}{\Gamma \vdash !\,t:\tau}$	Lesen einer Speicherzelle.
$:=\ \frac{\Gamma \vdash t_1:Cell\ \tau \quad \Gamma \vdash t_2:\tau}{\Gamma \vdash t_1 := t_2:unit}$	Schreiben auf eine Speicherzelle.
$\{\}I\ \frac{\Gamma \vdash t_1:\tau_1\ \ldots\ \Gamma \vdash t_n:\vdash \tau_n}{\Gamma \vdash \{m_1 = t_1, \ldots, m_n = t_n\}:\{m_1:\tau_1, \ldots \backslash m_n:\tau_n\}}$ *Typregel für Objekte als Records*	
$\{\}E\ \frac{\Gamma \vdash o:\{\ldots, m:\tau, \ldots\}}{\Gamma \vdash o.m:\tau}$	Typregel für Objekte als Records
$O1\ \frac{\sigma_1 \leq \tau_1 \ldots \sigma_n \leq \tau_n}{\{m_1:\sigma_1, \ldots, m_n;\sigma_n\} \leq \{m_1:\tau_1, \ldots, m_n:\tau_n\}}$ *Objektkonversion*	
$\boldsymbol{O2}\ \frac{}{\{\boldsymbol{m_1:\tau_1, \ldots, m_n:\tau_n, m_{n+1}:\tau_{n+1}, \ldots, m_{n+k}:\tau_{n+k}}\} \leq \{\boldsymbol{m_1:\tau_1, \ldots, m_n:\tau_n}\}}$ *Objekterweiterung*	
$\boldsymbol{SUB}\ \frac{\boldsymbol{\Gamma \vdash t:\sigma \quad \sigma \leq \tau}}{\boldsymbol{\Gamma \vdash t:\tau}}$	Sumsumption, Typkonformanz.

$\rightarrow SUB \dfrac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'}$ Subtyp-Beziehungen auf Funktionen. Kontravarianz im Parameter, Kovarianz im Ergebnis ***Dh. Parameter erben in die gegensätzliche Richtung, Rückgabetypen in dieselbe!***	
$refl \dfrac{}{\sigma \leq \sigma}$	Reflexiv. ZB für Trivialumformung {..}=Klasse
$transitiv \dfrac{\tau \leq \tau' \quad \tau' \leq \tau''}{\tau \leq \tau''}$	
$antisym \dfrac{\tau \leq \tau' \quad \tau' \leq \tau}{\tau = \tau'}$	
$\forall SUB \dfrac{\sigma' \leq \sigma \quad \text{für alle } p \leq \sigma' \text{ gilt } \tau[\alpha \leftarrow p] \leq \tau'[\alpha \leftarrow p]}{\forall \alpha <= \sigma.\tau <= \alpha <= \sigma'.\tau'}$ Also: Polymorphe Typen erben, wenn die Typparameter in dieselbe Richtung erben. Zwischen monomorphe und polymorphen Typen kann keine Erben-Beziehung bestehen.	

14 Erweiterungen des Typsystems

14.1 this-Zeiger für Methoden

TODO Typregel F270 FG406

Ändere Typregel für Objekte {}I:

Typregel mit Bezugsobjekt

$$\{\}I2 \frac{\Gamma' \vdash t_1 : \tau_1 \quad \dots \quad \Gamma' \vdash t_n : \tau_n}{\Gamma \vdash \{ m_1 = t_1, \dots, m_n = t_n \} : \{ m_1 : \tau_1, \dots, m_n : \tau_n \}}$$

wobei $\Gamma' = \Gamma, \texttt{this} : \{ m_1 : \tau_1, \dots, m_n : \tau_n \}$

Beispiel:

```
class A { int x = 42; int get() { return this.x; } }
```

$$\frac{\dots \quad \dfrac{\dfrac{}{\Gamma, \texttt{this} : \tau \vdash \texttt{this} : \tau}}{\Gamma, \texttt{this} : \tau \vdash \texttt{this.x} : \texttt{Cell int}}\{\}E}{\Gamma \vdash \{ \texttt{x} = \texttt{new } 42, \texttt{get} = !(\texttt{this.x}) \} : \underbrace{\{ \texttt{x} : \texttt{Cell int}, \texttt{get} : \texttt{int} \}}_{=\tau}}\{\}I2$$

Entscheidbarkeit der Typkorrektheit und Typinferenz nicht mehr trivial!

14.2 Polymorphe Typen

Generische Klassen entsprechen polymorphen Typen $\forall \alpha \leq \sigma.\tau$ („Alle Typen, die entstehen wenn in τ jedes α durch irgendeinen Typen $\sigma' \leq \sigma$ ersetzt wird.

Beispiel:

```
class G<P extends Point> {
  P move(P x, int i) { ... }
  P set(int i) { ... } }
class C<E> {
  <T> T f(T t, E e) { ... } }
```

$\tau_G = \forall \alpha \leq Point.\{move: \alpha \rightarrow int \rightarrow \alpha, set: int \rightarrow \alpha\}$

$$\tau_C = \forall \alpha.\{f: \forall \beta.\beta \rightarrow \alpha \rightarrow \beta\}$$

Abkürzung: $\forall \alpha.\tau = \forall \alpha \leq \tau_{Objekt}.\tau$

14.2.1 Instanziierung

TODO Instanziierungsregel F274

Beispiel:

```
class SP extends Point { ... }      C<Integer> c = new C<Integer>();
G<SP> g = new G<SP>();              String s = c.<String>f("a", null);
```

$$\tau_G = \forall \alpha \leq \texttt{Point}.\{\texttt{move} : \alpha \rightarrow \texttt{int} \rightarrow \alpha, \texttt{set} : \texttt{int} \rightarrow \alpha\}$$
$$\tau_C = \forall \alpha.\{\texttt{f} : \forall \beta.\beta \rightarrow \alpha \rightarrow \beta\}$$
$$\vdash !g : \tau_G<\tau_{SP}> = \{\texttt{move} : \tau_{SP} \rightarrow \texttt{int} \rightarrow \tau_{SP}, \texttt{set} : \texttt{int} \rightarrow \tau_{SP}\}$$
$$\vdash !c : \tau_C<\tau_{Integer}> = \{\texttt{f} : \underbrace{\forall \beta.\beta \rightarrow \tau_{Integer} \rightarrow \beta}_{=:\tau}\}$$
$$\vdash !c.f<\tau_{String}> : \tau<\texttt{String}> = \tau_{String} \rightarrow \tau_{Integer} \rightarrow \tau_{String}$$

14.2.2 Vererbung bei Instanzen generischer Klassen

Es gelten dieselben Einschränkungen wie bei sonstigen Vererbungen. **Es kann zwischen generischen Instanzen keine Vererbung geben! (TODO?)**

14.2.3 Polymorphe Subtypen

TODO Subtypenregel F281 und Beispiel

14.3 Rekursive Typen

Typkonstruktor eines rekursiv verwendeten Members in Klasse (zB `Node` Klasse mit member `next` vom Typ `Node`) löst zu unendlichem Typen auf.

Lösung: Syntax für rekursive Typen $\mu\, \alpha.F(\alpha)$ mit $\mu\, \alpha.F(\alpha) = F(\mu\, \alpha.F(\alpha))$. Damit ist $\tau_{Node} = \mu\, \alpha.F_{Node}(\alpha) = \mu\, \alpha.\{next: \alpha\}$.

Um unendliche Syntaxbäume zu vermeiden, konstruiert $\mu\, \alpha.F(\alpha)$ nur reguläre zyklische Bäume (Rückreferenz).

TODO Subtypregel F284

Beispiele siehe F.284ff (TODO min ein Beispiel hier)

14.4 Abstrakte Klasse

TODO

15 Inhalt

1 Einfachvererbung 1
1.1 Dynamische Bindung 1
1.1.1 Upcasts 1
1.1.2 Objektlayout und Subobjekte 1
1.2 Methodentabelle (vtable) 1
1.3 Type Casts 1
1.3.1 Upcast 1
1.3.2 Downcast 2
1.3.3 Statische & dynamische Variablenbindung 2
2 Softwaretechnische Aspekte 2
2.1 OO vs imperative Programmierung 2
3 Tücken der dynamischen Bindung 2
4 Invarianten und sichere Vererbung 2
5 Mehrfachvererbung 3
5.1 Nicht-virtuelle Vererbung in C++ 3
5.2 Virtuelle Vererbung in C++ 3
5.3 Subobjektgraphen 3
5.3.1 Konstruktion mittels Induktion 3
5.3.2 Static Lookup 3
6 Implementierung von Mehrfachvererbung 4
6.1 C++ Typecasts 4
6.1.1 Nicht-virtuelle Einfachvererbung 4
6.1.2 Nicht-virtuelle Mehrfachvererbung 4
6.1.3 Virtuelle Mehrfachvererbung 4
6.2 Vererbung mit vtables 5
6.2.1 Einfachvererbung mit vtables 5
6.2.2 Mehrfachvererbung mit vtables 5
7 Java Interfaces 6
7.1 Implementieren von Interfaces 6
7.1.1 Ansatz: C++-Strategie 6
7.1.2 Problem: Dynamisches Laden von Klassen/Interfaces 6
7.1.3 Interface Method Tables 6
7.2 Java: Default-Methoden 6
8 Überladungen 6
8.1 Spezifischste Methode 6
8.2 Überladung und dynamische Bindung 6
8.3 Smart Pointer 6
9 Innere Klassen 7
9.1 Statische innere Klasse 7
9.2 Dynamische innere Klasse 7
9.3 Anonyme Klassen 7
9.4 Beispiel 7
10 Generics 7
11 Tyrannei der dominanten Dekomposition 8
11.1 Ansätze 8
11.1.1 Datentypen als Klassenhierarchie, Operationen als dynamisch gebundene Methoden 8
11.1.2 Visitor-Pattern 8
11.2 Problem 8
11.3 Lösungsansätze 8
11.3.1 Multimethoden 8
11.3.2 Traits, Mixins, abstrakte Typmember 9
11.3.3 Virtuelle Klassen 9
12 Programmanalyse 9
12.1 Eigenschaften 9
12.1.1 Fluss-Sensitivität 10
12.1.2 Kontext-Sensitivität 10
12.2 Rapid Type Analysis 10
12.2.1 Call-Graph 10
12.2.2 Reduktion der Call-Graph-Größe 10
12.2.3 RTA als Constraint Problem 11
12.2.4 Fazit 11
12.3 Points-To-Analysis 11
12.3.1 Points-To-Graph 11
12.3.2 Zuweisung nach Andersen 11
12.3.3 Lösung des Mengengleichungssystems 11
12.3.4 Zugriffe auf Attribute 12
12.3.5 Systeme mit bedingten Ungleichungen 12
13 Typsysteme für Objektorientierung 13
13.1 Grundlagen 13
13.1.1 Regelsysteme 13

13.1.2 Lambda-Kalkül 13
13.2 Objekte als einfache Records 13
13.3 Typkonversionen 13
13.3.1 Typkonversionen für Objekte 13
13.3.2 Typkonversionen für Methoden 14
13.4 Ko- und Kontravarianz bei Vererbung 14
13.5 Typkonstruktor 14
13.6 Array-Anomalie in Java 14
14 Erweiterungen des Typsystems 15
14.1 this-Zeiger für Methoden 15
14.2 Polymorphe Typen 15
14.2.1 Instanziierung 15
14.2.2 Vererbung bei Instanzen generischer Klassen 15
14.2.3 Polymorphe Subtypen 15
14.3 Rekursive Typen 15
14.4 Abstrakte Klasse 16
15 Inhalt 17

16 Anhang

16.1 Generics Stackoverflow Comment

https://stackoverflow.com/a/4343547/2692307

16.1.1 `extends`

The wildcard declaration of `List<? extends Number> foo3` means that any of these are legal assignments:

```
List<? extends Number> foo3 = new ArrayList<Number>(); // Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>(); // Double extends Number
```

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to read from `List foo3`:
   - You can read a **`Number`** because any of the lists that could be assigned to `foo3` contain a `Number` or a subclass of `Number`.
   - You can't read an `Integer` because `foo3` could be pointing at a `List<Double>`.
   - You can't read a `Double` because `foo3` could be pointing at a `List<Integer>`.
2. **Writing** - Given the above possible assignments, what type of object could you add to `List foo3` that would be legal for **all** the above possible `ArrayList` assignments:
   - You can't add an `Integer` because `foo3` could be pointing at a `List<Double>`.
   - You can't add a `Double` because `foo3` could be pointing at a `List<Integer>`.
   - You can't add a `Number` because `foo3` could be pointing at a `List<Integer>`.

You can't add any object to `List<? extends T>` because you can't guarantee what kind of `List` it is really pointing to, so you can't guarantee that the object is allowed in that `List`. The only "guarantee" is that you can only read from it and you'll get a `T` or subclass of `T`.

16.1.2 `super`

Now consider `List <? super T>`.

The wildcard declaration of `List<? super Integer> foo3` means that any of these are legal assignments:

```
List<? super Integer> foo3 = new ArrayList<Integer>(); // Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();  // Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();  // Object is a superclass of Integer
```

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to receive when you read from `List foo3`:
   - You aren't guaranteed an `Integer` because `foo3` could be pointing at a `List<Number>` or `List<Object>`.
   - You aren't guaranteed a `Number` because `foo3` could be pointing at a `List<Object>`.
   - The **only** guarantee is that you will get an instance of an **`Object`** or subclass of `Object` (but you don't know what subclass).
2. **Writing** - Given the above possible assignments, what type of object could you add to `List foo3` that would be legal for **all** the above possible `ArrayList` assignments:
   - You can add an `Integer` because an `Integer` is allowed in any of above lists.
   - You can add an instance of a subclass of `Integer` because an instance of a subclass of `Integer` is allowed in any of the above lists.
   - You can't add a `Double` because `foo3` could be pointing at an `ArrayList<Integer>`.
   - You can't add a `Number` because `foo3` could be pointing at an `ArrayList<Integer>`.
   - You can't add an `Object` because `foo3` could be pointing at an `ArrayList<Integer>`.

16.1.3 PECS

Remember *PECS*: **"Producer Extends, Consumer Super"**.

- **"Producer Extends"** - If you need a `List` to produce `T` values (you want to read `T`s from the list), you need to declare it with `? extends T`, e.g. `List<? extends Integer>`. But you cannot add to this list.
- **"Consumer Super"** - If you need a `List` to consume `T` values (you want to write `T`s into the list), you need to declare it with `? super T`, e.g. `List<? super Integer>`. But there are no guarantees what type of object you may read from this list.
- If you need to both read from and write to a list, you need to declare it exactly with no wildcards, e.g. `List<Integer>`.

16.1.4 Example

Note this example from the Java Generics FAQ. Note how the source list `src` (the producing list) uses `extends`, and the destination list `dest` (the consuming list) uses `super`:

```
public class Collections {
  public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    for (int i = 0; i < src.size(); i++)
      dest.set(i, src.get(i));
  }
}
```

Also see How can I add to List<? extends Number> data structures?

16.2 Fazit

`? extends C` im Typparameter: Der Typ liegt unterhalb C. Ich kann nur davon *als C lesen* und *Object-Typen darin schreiben*. Schreiben von C-Typen ist nicht möglich.

`? super C` im Typparameter: Der Typ liegt oberhalb C. Ich kann davon nur *als Object lesen*, nicht als C, dafür *C-Objekte darin schreiben*.