
Heute: Vererbung

- Konzept: Vererbung und Polymorphie
- Typen: Superklassen, Polymorphie
- Laufzeitverhalten
 - Erweiterung des Instanz-Structs
 - Überschreiben von Methoden
- Dynamische Typinformation: Downcast & instanceof

Ein einfaches Beispiel

```
class A {
    int i;
    void e() { System.out.println("A.e"); }
    void f() { System.out.println("A.f"); }
}
class B extends A {
    int j;
    void f() { System.out.println("B.f"); }
    void g() { System.out.println("B does have g()"); }
}
class User {
    void workWith(A a) {
        a.f();
        if (a instanceof B) {
            B b = (B)a;
            b.g();
        }
    }
}
```

Vererbung in der JLS

§8.1.3: The optional extends clause in a class declaration specifies the [direct superclass](#) of the current class. A class is said to be a [direct subclass](#) of the class it extends. The direct superclass is the class from whose implementation [the implementation of the current class is derived](#). [. . .] If the class declaration for any [..] class [other than Object] has no extends clause, then the class has the class Object as its implicit direct superclass.

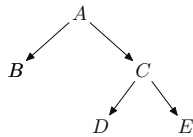
- Ziel: Implementierung von der Implementierung der Superklasse ableiten
- ⇒ OO-Programmierung: Wiederverwendung von Code
- Jede Klasse hat eine direkte Superklasse, mindestens Object
- ⇒ Object ist die Wurzel der Klassenhierarchie in Java.

Vererbung in der JLS

§8.2: The members of a class type are all of the following:

- Members inherited from its direct superclass (§8.1.3), except in class Object, which has no direct superclass [. . .]
 - Members declared in the body of the class (§8.1.5)
- [. . .] [Constructors](#) [. . .] are not members and therefore [are not inherited](#).
- Vererbung ist konzeptuell die Übernahme von Feldern und Methoden
 - In Spezifikation: Klassen sind völlig eigenständig ≠ Implementierung
 - Die neue Klasse verhält sich zunächst wie ihre direkte Superklasse
 - Sie kann jedoch neue Felder und Methoden hinzufügen
 - Konstruktoren sind spezifisch für jede Klasse, sie werden nicht vererbt

Vererbungshierarchie



- Vererbung kann fortgesetzt werden
- ⇒ Allgemeine Superklassen / Subklassen
- ⇒ Baum mit Klassen als Knoten
 - Kinder sind abgeleitete Klassen
 - Vater ist Superklasse
- Erweiterung Mehrfachvererbung: Erben von mehreren Klassen
→ Gerichter azyklischer Graph → Nächste Stunde

Erinnerung: Konversionen (JLS §5.1)

- Eine Konversion wandelt einen Wert v vom Typ s um in einen Wert v' mit Typ t .
- Eine solche Konversion kann ausgelöst werden durch
 - Den Kontext, in dem v verwendet wird (beispielsweise Zuweisung)
 - Einen expliziten Cast-Ausdruck $(t)e$
- Die JLS sieht folgende Arten der Konversion vor
 - Identity conversion
 - Widening Primitive Conversion (\approx mehr Bits für die Darstellung)
 - Narrowing Primitive Conversions (\approx weniger Bits für die Darstellung)
 - Widening Reference Conversions
 - Narrowing Reference
 - String Conversions: Umwandlung in einen String

Eigenschaften nach Programmiererfahrung

- B-Instanzen können “mehr” als A-Instanzen
 - Sie verlieren keine Eigenschaften
 - Legale Methodenaufrufe für A-Instanzen sind auch legal für B-Instanzen
- B-Instanzen sind “Spezialfälle” und dürfen sich im Detail anders verhalten
- Eine B-Instanz kann in einer A-Variable gespeichert sein
- Nachfolgend: Konzepte
 - Polymorphie
 - Konversionen
 - Typregeln
- Dann: Implementierung

Widening Reference Conversions (JLS §5.1.4)

The following conversions are called the widening reference conversions:

- From any class type S to any class type T , provided that S is a subclass of T .
- From the null type to any class type $[\dots]$
- $[\dots$ Fälle für Interfaces und Arrays]

Such conversions never require a special action at run time $[\dots]$. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

- Referenzen auf Instanzen einer Klasse können in Referenzen ihrer Superklassen konvertiert werden
- Diese Konversion ändert die Referenz (den konvertierten Wert) nicht

Tc: Konversionen berechnen

Tc

```
let conv a b =
  match (a,b) with
  | Ty_null, Ty_ref b -> Conv_none
  | Ty_ref(a), Ty_ref(b) ->
    if a == b
    then Conv_none
    else if has_super_type a b
    then Conv_up(a,b)
    else type_error (string_of_ref_ty a ^
                     " is not a subtype of " ^
                     string_of_ref_ty b)
```

- Erinnerung: conv berechnet alle notwendige Konversionen
- In Tc: set_exp_conv e hält Konversionen für Ausdrücke
- In Cg: patch_conv_to_exp e konvertiert Ergebnis

Ersetzbarkeit

- Gängige Formulierung von Konversion ohne Wertänderung
- Typregel:
Wenn der Typ von v eine Klasse C ist und C' die Superklasse von C ist, dann ist auch C' ein Typ von v .
- Knapper ausgedrückt (waagerechter Strich ist "folgt"; \leq als "ist Subklasse von")

$$\frac{v : s \quad s \leq t}{v : t}$$

⇒ Instanzen von C können immer für Instanzen von C' eingesetzt werden (*substitutability*)

- Nur der Typ wird geändert, nicht der Wert

Cg: Konversionen durchführen

Cg

```
let patch_conv_to_exp conv exp =
  match conv with
  | Conv_up(Rty_cls _, Rty_cls _) -> exp
```

- Eine widening conversion zwischen Klassen hat keinen Effekt zur Laufzeit

⇒ JLS implementiert

Polymorphie

- Eine konkrete Instanz v von C hat potentiell mehrere Typen (alle Superklassen von C)
- Eine Variable von Klassentyp C kann Instanzen aller Subklassen von C aufnehmen
- Jede Referenz r auf ein Objekt hat zwei Typen
 - Der **dynamische Typ** ist die Klasse, von der das Objekt, auf das r verweist, eine Instanz ist.
 - Der **statische Typ** von r wird vom Compiler berechnet (z.B. aus einer Deklaration entnommen)
- Korrektheit: Der statische Typ ist immer eine Superklasse des dynamischen Typs.

Überschreiben von Methoden

§8.4.6.1 (Overriding by Instance Methods) An instance method m_1 declared in a class C overrides another method **with the same signature**, m_2 , declared in [a super-]class A [of C] if m_2 is [...] accessible from C [...]

§8.4.6.3 (Requirements in Overriding) If a method declaration overrides or hides the declaration of another method, then a compile-time error occurs if they have **different return types** [...].

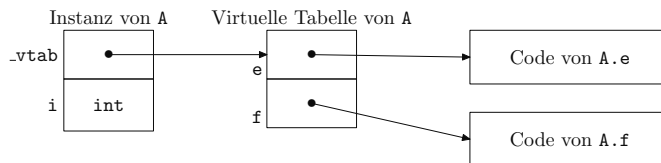
- Definition des Begriffs Überschreiben (method overriding)
- Detail: Die Rückgabetypen gehören nicht zur Signatur
- Noch keine Aussage zur Bedeutung
- Idee: Überschreiben ersetzt den Code der Methode

Laufzeit-Datenstrukturen

```
A a = new B()
```

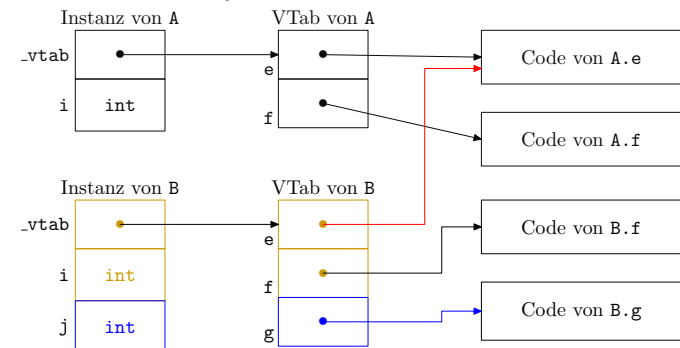
- Bei Zugriff auf a weiss der Compiler (im allgemeinen) nicht, daß sich dahinter eine Instanz von B verbirgt
- ⇒ B-Instanzen müssen zur Laufzeit als A-Instanzen behandelt werden können
- ⇒ B-Instanzen fügen zu Instanz-Struct und Virtueller Tabelle am Ende neue Einträge hinzu, die Struktur des vorderen Teils bleibt unverändert.

Erinnerung: Virtuelle Tabelle



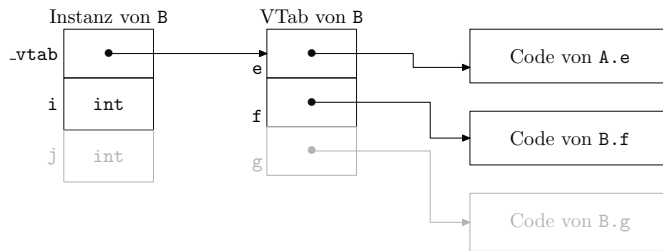
- Objekte enthalten den Code ihrer Methoden
- ⇒ Funktionszeiger
- Funktionszeiger sind für alle Instanzen einer Klasse gleich
- ⇒ Auslagern in virtuelle Tabelle

Beispiel: Klassen A und B



- Vorderer Teil von Instanz- und virtueller Tabelle haben gleichen Aufbau → Zugriff auf `B.i`, `B.e()`, `B.f()` genau wie auf `A.i`, `B.e()`, `B.f()`
- B fügt neues Feld und neue Methode hinzu
- B erbt Eintrag und Code von A.e()
- B überschreibt `f()`: Aufruf `a->_vtab->f()` führt neuen Code aus

Widening Reference Konversion



- Vorstellung: Eine B-Instanz, betrachtet als A-Instanz
 - Der statische Typ A beschreibt die schwarzen Anteile
- ⇒ B-Instanz ist wie A-Instanz verwendbar
- ⇒ Feld j und Methode g des dynamischen Typs sind nicht sichtbar

Tc: Methodenaufruf

```
Exp_call(e',id,args) ->
  let ty_e' = chk_exp e' in
  let _ = map chk_exp args in
  let (formals,ret) =
    (match ty_e' with
     | (Ty_ref (Rty_cls cls)) ->
       let m = find_called_cls_meth cls id args in
       set_exp_acc_meth e m;
       (m.meth_formals, m.meth_ret))
  in
  chk_args_against_formals args formals;
  ret
```

- Verwende den Typ von e' um die verfügbaren Methoden zu finden
- Wähle die passendste Methode aus (und setze Property exp_acc_meth) (→ Überladung, noch nicht implementiert)
- Prüfe Argumente für die ausgewählte Methode
- Ergebnistyp ist deklarierter Rückgabety

Cg: Methodenaufruf

```
| Exp_call(e',id,args) ->
  let exp' = cg_exp e' in
  let args' = map cg_exp args in
  let argtmp' = map (fun _ -> Temps.alloc()) args in
  let argass = map2
    (fun t a -> C.Exp_assign(t, a))
    argtmp' args' in
  let meth = exp_acc_meth exp in
  ...
```

- Werte e' und args aus und reserviere dabei temporäre Variablen für die Ergebnisse dieser Ausdrücke
 - Typchecker hatte aufzurufende Methode exp_acc_meth bestimmt
- ⇒ Alle Daten für Codegenerierung vorhanden

Cg: Statischer Methodenaufruf

```
match exp_ty e' with
  Ty_ref(Rty_cls cls) ->
    if is_static_meth meth
    then C.Exp_seq
      (argass @ [C.Exp_call(C.Exp_var (meth_fun meth), argtmp')])
    else ...
```

- Speichere Argumente in temporären Variablen → Auswertungsreihenfolge
 - Wenn die ausgewählte Methode statisch ist
 - Es ist klar, welcher Code ausgeführt wird (!)
 - Der Aufruf hat kein this Argument
 - Genau ein Funktionsaufruf in C
 - Der Methodenname steht fest
 - Genau die Argumente werden übergeben
- ✓ Ergebnis: Rückgabe der Funktion, wie von Tc berechnet

Cg: Dynamischer Methodenaufruf

```
let inst = cls_inst cls in
  C.Exp_seq(
    [ C.Exp_assign(this, exp') ] @ argass @
    [ C.Exp_call(
      C.Exp_deref_acc(C.Exp_deref_acc(
        C.Exp_cast(C.Ty_ptr(C.Ty_struct_ref(inst.inst_decl_nm)),
          this),
          "_vtab"), meth_nm meth),
        this :: argtmp') ])
```

- Speichere in this das Ergebnis von exp'
- Typcheck garantiert: this ist Zeiger auf Instanz von cls = exp_ty e'
- ⇒ C-Cast in C.Exp_cast(...) auf Instanz-Struct dieser Klasse erlaubt
- ⇒ Zugriff auf virtuelle Tabelle der Instanz erlaubt
- ⇒ Funktionszeiger aus virtueller Tabelle der Instanz
- ✓ Ergebnis: Rückgabe der Funktion, wie von Tc berechnet

Layout: Virtuelle Tabelle

Layout

```
let create_cls_vtab cls =
  let rec entries_for cls =
    ...
  in
  {
    vtab_decl_nm = cls_nm cls ^ nm_sep ^ "vtab";
    vtab_has_off = false;
    vtab_def_nm = Some(cls_nm cls ^ nm_sep ^ "vtab");
    vtab_entries = entries_for cls;
  }
```

- Hauptarbeit: Einträge aller Methoden aufsammeln
- entries_for definiert direkt Übernahme der Einträge aus Superklassen

Beispielaufruf

```
System.out.println_int(p.getX());

((__tmp_0)=(java$lang$System_out),
 (__tmp_2)=(((__tmp_1)=(p$6),
              (((struct Point$Obj*)(__tmp_1))->_vtab)
               ->getX__I)(__tmp_1))),
 (((struct java$lang$PrintStream$Obj*)(__tmp_0))->_vtab)
 ->println_int_I_V)(__tmp_0,__tmp_2));
```

Layout: Überschriebene Methode finden

Layout

```
let defined_cls_meth m cls =
  let rec find_in cls =
    try Some(find (equ_meth_head m) cls.cls_meths)
    with Not_found ->
      if has_extends cls
      then find_in cls.cls_extends
      else None
  in find_in cls
```

```
let overridden_cls_meth m cls =
  if has_extends cls
  then defined_cls_meth m cls.cls_extends
  else None
```

- defined_cls_meth
 - Finde eine Methodendeklaration mit gleicher Signatur wie m
 - Durchsuche falls nötig die Superklassen
- overridden_cls_meth: Finde Definition der Methode in Superklasse

Layout: Methoden zusammenstellen

Layout

```
let rec entries_for cls =
  let super_entries =
    (if has_extends cls
     then (copy_vtab (cls_vtab cls.cls_extends)).vtab_entries
     else [])
  in let new_meths =
    mapfilter
      (fun m -> match overridden_cls_meth m cls with
        None -> Some (Vtab_meth { (* neue Methode *)
                                   vtm_decl = m;
                                   vtm_def  = Some m;
                                 })
        | Some m' -> (* Überschrieben Methode *)
          let vtm = entry_for_meth_decl m' super_entries
          in vtm.vtm_def <- Some m; (* override *)
            None)
      cls.cls_meths
  in
  super_entries @ new_meths
```

Methodenaufruf in der JLS

- Die virtuelle Tabelle ist eine Implementierungstechnik
 - Sehr detailliert: Was passiert genau zur Laufzeit?
 - Im Vordergrund: **Wie** wird eine Methode gefunden
 - Im Hintergrund: **Welche** Methode wird eigentlich gefunden?
- Spezifikation (JLS §15.12.4): Welche Methode soll gefunden werden?
- Konzept **dynamic dispatch**: Gegeben ist ein Objekt v und eine Methode f_S mit Signatur S . Finde **zur Laufzeit** die Methode f_S in v und rufe diese auf.
- Fragen:
 - Was passiert, wenn mehrere Methoden gefunden werden?
 - Was passiert, wenn keine Methode gefunden wird?

Runtime Evaluation of Method Invocation §15.12.4

§15.12.4.4 (Locate Method to Invoke) [. . .] an instance method is to be invoked and there is a **target reference**. [. . .] the target reference is said to refer to a **target object**.

A **dynamic method lookup** is used. The dynamic lookup process starts from a class S , [which] is initially the **actual run-time class** R of the target object.

- Beim normalen (virtuellen) Methodenaufruf wird die aufgerufene Methode zur Laufzeit ermittelt
- Methoden werden aus Klassen entnommen
- Die Suche startet im dynamischen Typ des Zielobjektes
- Hinweis: Suche verwendet **Deskriptor** (= Name+Parametertypen+Rückgabe), nicht die Signatur (=Name+Parametertypen)

Runtime Evaluation of Method Invocation §15.12.4

Let X be the **compile-time type** of the target reference of the method invocation.

1. If class S contains a declaration for a method named m with the **same descriptor** [. . .] and the declaration in S overrides $X.m$, then the method declared in S is the method to be invoked, and the procedure terminates.
 2. Otherwise, if S has a superclass, this same lookup procedure is performed recursively using the direct superclass of S in place of S
- Typchecker hatte die Methode im statischen Typ X nachgeschlagen
 - Compiler hat Typ X im Code für Aufruf vermerkt
 - Bei uns: Der Cast auf die passende Instanz
 - In Java: `invokevirtual` Instruktion hat Typargument
 - Durchsuche alle Superklassen, um die Methode zu finden, die $X.f$ überschreibt (und damit den Code für diese Methode liefert)

Typchecker schliesst wieder Fehler aus!

- Eine Methode, die $X.f$ überschreibt, wird sicher gefunden
 - Der Typchecker durchsucht die Superklassen genau parallel
 - Er hätte eine Fehlermeldung gebracht, wenn er keine Methode gefunden hätte
 - Es nie mehr als eine Methode gefunden
 - Innerhalb einer Klasse kann eine Methode nur einmal definiert werden
 - Bei Überschreiben wird die Methode genommen, die zuerst gefunden wird, also werden Methoden aus Subklassen bevorzugt
- ✓ Typchecker hat sichergestellt, dass Laufzeitsuche erfolgreich ist

Der Instanceof-Operator

```
void workWith(A a) {
    if (a instanceof B) {
        System.out.println("you passed me a B instance\n");
    }
}
```

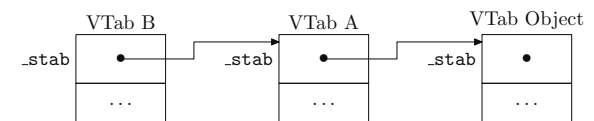
- Methodenaufruf gerade so gemacht, dass er unabhängig vom dynamischen Typ funktioniert.
- ⇒ Zur Laufzeit wird keine Typinformation benötigt
- Gegenteil instanceof: Wir brauchen Typinformation zur Laufzeit
 - Typinformationen sind Datenstrukturen
 - Diese werden vom Compiler angelegt
 - Müssen zur Laufzeit vom Objekt aus erreichbar sein
 - C++ Terminologie: [runtime type information](#) (RTTI)

Vergleich Virtuelle Tabellen

- Beziehung Spezifikation ↔ Implementierung
 - Spezifikation: Suche vom dynamischen Typ aufwärts in der Klassenhierarchie
 - Die Virtuelle Tabelle wird in der Klassenhierarchie abwärts konstruiert
 - Korrektheit: Die bei der Suche gefundene Methode ist gerade die in der Virtuellen Tabelle eingetragene!
 - Die erste bei der Suche gefundene Methode ist gerade die zuletzt in die virtuelle Tabelle des dynamischen Typs eingetragene.
 - Der Methodendeskriptor gibt an, an welcher Stelle in der Virtuellen Tabelle die Methode abgelegt wurde.
- ⇒ Der Compiler kann X und den Methodendeskriptor von m vorausberechnen

Lösung 1: Super-Zeiger

- `_vtab`-Zeiger zeigen schon auf "Laufzeittypinformation": Die Virtuelle Tabelle in `_vtab` identifiziert eindeutig die Klasse, aus der ein Objekt instanziiert wurde.
- Idee: Erweitere virtuelle Tabellen um Zeiger auf Tabelle der Superklasse.



- $e \text{ instanceof } C$: Suche ab $e \rightarrow _vtab$ nach $C\$vtab$
- ✗ Ineffizient: $O(n)$, wenn der dynamische Typ von e in n Schritten von Object abgeleitet ist.

Lösung 2: Subklassen-Tabelle

- Betrachte Relation $\leq := \{(C, C') \mid C \text{ ist Subklasse von } C'\}$
 - Die Anzahl der Klassen C, C' ist endlich, also ist auch \leq endlich
- ⇒ Darstellbar als Bittabelle SUB
- Ordne jeder Klasse einen eindeutigen Index i_C zu und speichere ihn in virtueller Tabelle von C
 - Compiler füllt die Tabelle SUB, so aus daß

$$\text{SUB}(i_C, i_{C'}) = 1 \text{ genau dann wenn } C \leq C'$$

- ✓ Laufzeit $e \text{ instanceof } C$ ist $O(1)$: Index aus virtuellen Tabellen von e und C entnehmen und zugreifen
- ✗ Speicherplatz $O(n^2)$ wobei n die Anzahl der Klassen im Programm ist
- ✗ Die allermeisten Einträge in der Tabelle sind 0

Umsetzung im Compiler

- Erweitere `Rt_tables` um Typ `rtti_cls` für die RTTI-Tabellen

```
type rtti_cls = {
  rtti_cls_def_nm  : string;
  rtti_cls_decl_nm : string;
  rtti_cls_name    : string;
  rtti_cls_level   : int;
  rtti_cls_chain   : rtti_cls list;
}
```

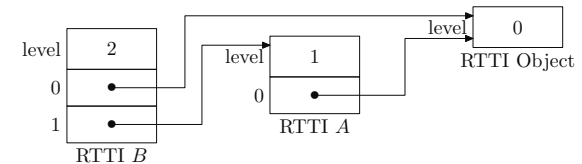
- Erweitere `vtab` um Eintrag für zugehörige `rtti_cls`

```
type vtab = {
  ...
  mutable vtab_rtti : rtti;
}
```

- Codegenerator erzeugt Struct-Deklarationen & -definitionen

Lösung 3: Array mit Superklassen

- Ebene (Level) einer Klasse ist Anzahl der `extends` bis Object
- Speichere in der Tabelle `rtti` für jede Klasse einen Vektor mit Zeigern auf `rtti`-Tabellen aller Superklassen ab



- Virtuelle Tabelle enthält Zeiger auf entsprechende `rtti`-Tabelle
- $e \text{ instanceof } C$ genau dann wenn
`C$rtti->level < e->_vtab->rtti->level &&`
`e->_vtab->rtti->tab[C$rtti->level] == &C$rtti`

Cg: Instanceof-Ausdrücke

```
let rec cg_exp exp =
  ...
  | Exp_instanceof(e, ty) ->
    let tmp = Temps.alloc (exp_ty e)
    in let rtti =
      (match ty with
       Ty_ref(Rty_cls cls) -> cls_rtti cls
       | _ -> assert false)
    in C.Exp_seq [
      C.Exp_call(Rt.instance_of_fun,
                [ cg_exp e;
                  C.Exp_addr(C.Exp_var rtti.rtti_cls_de

```

- Bestimmung der Ziel-RTTI-Tabelle
- Hauptarbeit in Hilfsfunktion `rt_instanceof`
- Bemerkung: Hätten auch inlinen können

Hilfsfunktion `rt_instanceof`

```
int rt_instanceof(void *obj, void *target_rtti) {
    int target_level = ((struct java$lang$Object$rtti*)target_rtti)->
    struct java$lang$Object$rtti *obj_rtti = ((struct java$lang$Object$
    if (obj_rtti == target_rtti) return 1;
    return target_level < obj_rtti->level &
        obj_rtti->chain[target_level] == target_rtti;
}
```

- Der dynamische Typ von `obj` ist nicht selbst in `chain` enthalten
→ Fallunterscheidung
- Danach: Hergeleiteten Ausdruck verwenden

Zusammenfassung Vererbung

- Ziel: Wiederverwendung von Code
- Polymorphie
- Dynamischer Methodenaufruf
 - Spezifikation: Laufzeitsuche
 - Implementierung: Virtuelle Tabellen
- Downcast & `instanceof`
 - Laufzeittypinformation
 - Realisierung in $O(1)$

Downcasts

- Cast $(C')e$
 - Compiler berechnet Typ C von e
 - Statischer Typ des Gesamtausdrucks ist C'⇒ Forderung: Dynamischer Typ von e muß Subklasse von C' sein
- Fall $C \leq C'$: Typchecker weiss schon, daß Forderung erfüllt ist
- Fall $C' \leq C$: Forderung möglicherweise erfüllt → Laufzeitcheck

```
void *rt_downcast(void *obj, void *target_rtti) {
    if (!rt_instanceof(obj, target_rtti)) {
        fprintf(stderr, "Cast from %s to %s fails at run-time\n",
            ((struct java$lang$Object$obj*)obj)->_vtab->_rtti->na
            ((struct java$lang$Object$rtti*)target_rtti)->name);
        fflush(stderr);
        exit(1);
    }
    return obj;
}
```