Softwaretechnik

Vorlesung 04: Featherweight Java

Peter Thiemann

Universität Freiburg, Germany

SS 2008

Inhalt

Typsicherheit von Java

- 1995 öffentliche Vorstellung von Java
- erlangt schnell große Popularität
- Fragen
  - Typsicherheit?
  - Was bedeutet das für Java?
- Erst 1997/98 geklärt
  - Drossopoulou/Eisenbach
  - Flatt/Krishnamurthi/Felleisen
  - Igarashi/Pierce/Wadler (Featherweight Java, FJ)

Featherweight Java

- Konstruktion eines formalen Modells:
  Abwägung zwischen Vollständigkeit und Kompaktheit
- FJ: minimales Modell (Kompaktheit)
- Komplette Definition: eine Seite
- Ziel:
  - die wichtigsten Sprachmerkmale
  - möglichst kurzer Beweis für Typsicherheit
  - FJ $\subseteq$ Java

Die Sprache FJ

- Klassendefinition
- Objekterzeugung **new**
- Methodenaufruf (*dynamic dispatch*), Rekursion mit **this**
- Feldzugriff
- Typcast
- Überschreiben von Methoden (*override*)
- Subtypen

Auslassungen

- Zuweisungen
- Interfaces
- Überladung (*overload*)
- **super**-Aufrufe
- **null**-Zeiger
- primitive Typen
- abstrakte Methoden
- innere Klassen
- Überdecken von Feldern der Superklasse
- Zugriffskontrolle (**private**, **public**, **protected**)
- Ausnahmen (*exceptions*)
- Nebenläufigkeit
- Reflektion

Beispielprogramm

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

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

class Pair extends Object {
  Object fst;
  Object snd;
  // Constructor
  Pair (Object fst, Object snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  // Method definition
  Pair setfst (Object newfst) {
    return new Pair (newfst, this.snd);
  }
}
```

Erklärungen

- Klassendefinition: Superklasse wird immer angegeben
- Konstruktoren:
  - genau einer pro Klasse, immer angegeben
  - Argumente entsprechen genau den Felder
  - immer die gleiche Form: **super**-Aufruf, dann Kopieren der Argumente in ihre Felder
- Feldzugriffe und Methodenaufrufe **immer** mit Empfängerobjekt
- Methodenrumpf: immer die Form **return**...

Beispiele für Auswertung

Methodenaufruf

```
new Pair (new A(), new B()).setfst (new B())
// wird ausgewertet nach
new Pair (new B(), new B())
```

Beispiele für Auswertung

Methodenaufruf

```
new Pair (new A(), new B()).setfst (new B())
// wird ausgewertet nach
new Pair (new B(), new B())
```

Typcast

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

- enthält Typcast (Pair)
- erforderlich, weil **new** Pair (...).fst den Typ Object hat

Beispiele für Auswertung

Feldzugriff

```
new Pair (new A (), new B ()).snd
// wird ausgewertet nach
new B()
```

Beispiele für Auswertung

Feldzugriff

```
new Pair (new A (), new B ()).snd
// wird ausgewertet nach
new B()
```

Methodenaufruf

```
new Pair (new A(), new B()).setfst (new B())
```

liefert eine Substitution

$$[\textbf{new}\ \mathsf{B}()/\mathsf{newfst},\quad \textbf{new}\ \mathsf{Pair}\ (\textbf{new}\ \mathsf{A}(),\ \textbf{new}\ \mathsf{B}())/\textbf{this}]$$

unter der der Methodenrumpf **new** Pair (newfst, **this**.snd) ausgeführt wird. Ausführen der Substitution liefert

```
new Pair (new B(), new Pair (new A(), new B()).snd)
```

Beispiele für Auswertung

Typcast

```
(Pair)new Pair (new A (), new B ())
// wird ausgewertet nach
new Pair (new A (), new B ())
```

- Laufzeittest ob Pair ein Subtyp von Pair ist

Beispiele für Auswertung

Typcast

```
(Pair)new Pair (new A (), new B ())
// wird ausgewertet nach
new Pair (new A (), new B ())
```

- Laufzeittest ob Pair ein Subtyp von Pair ist

Call-by-Value Auswertung

```
((Pair) new Pair (new Pair (new A(), new B ()), new A()).fst).snd
// →
((Pair) new Pair (new A(), new B ())).snd
// →
new Pair (new A(), new B ()).snd
// →
new B()
```

Laufzeitfehler

Zugriff auf nicht-existierendes Feld

```
new A().fst
```

Kein Wert, keine Berechnungsregel greift

Laufzeitfehler

Zugriff auf nicht-existierendes Feld

```
new A().fst
```

Kein Wert, keine Berechnungsregel greift

Aufruf einer nicht-existierenden Methode

```
new A().setfst (new B())
```

Kein Wert, keine Berechnungsregel greift

Laufzeitfehler

Zugriff auf nicht-existierendes Feld

```
new A().fst
```

Kein Wert, keine Berechnungsregel greift

Aufruf einer nicht-existierenden Methode

```
new A().setfst (new B())
```

Kein Wert, keine Berechnungsregel greift

Illegaler Typcast

```
(B)new A ()
```

- A ist nicht Subtyp von B

$\Rightarrow$ Kein Wert, keine Berechnungsregel greift

Garantieen von Javas Typsystem

Wenn ein Java-Programm typkorrekt ist, dann

- treten **keine** Zugriffe auf nicht-existierende Felder auf,
- treten **keine** Aufrufe von nicht-existierenden Methoden auf, aber es
- können illegale Typcasts auftreten.

Formale Definition

Syntax

CL	::=		Klassendefinition
		class C **extends** $D\,\{C_1\ f_1; \ldots\ K\ M_1 \ldots\}$	
K	::=		Konstruktordefinition
		$C(C_1\ f_1, \ldots)\ \{$**super**$(g_1, \ldots);$ **this**$.f_1 = f_1; \ldots\}$	
M	::=		Methodendefinition
		$C\ m(C_1\ x_1, \ldots)\ \{$**return** $t;\}$	
t	::=		Ausdrücke/Terme
		x	Variable
		$t.f$	Feldzugriff
		$t.m(t_1, \ldots)$	Methodenaufruf
		new $C(t_1, \ldots)$	Objekterzeugung
		$(C)\ t$	Typcast
v	::=		Werte
		new $C(v_1, \ldots)$	Objekterzeugung

Syntax—Konventionen

- **this**
  - spezielle Variable, nicht als Feldname oder Parameter
  - implizit gebunden in jedem Methodenrumpf
- Sequenzen von Feldnamen, Parameternamen und Methodennamen enthalten keine Wiederholung
- **class** C **extends** $D\ \{C_1\ f_1; \ldots\ K\ M_1 \ldots\}$
  - definiert Klasse C als Subklasse von D
  - Felder $f_1 \ldots$ mit Typen $C_1 \ldots$
  - Konstruktor K
  - Methoden $M_1 \ldots$
  - Felder werden zu den Feldern von D hinzugefügt, Verdeckung nicht erlaubt

Syntax—Konventionen

- $C(D_1\ g_1, \ldots, C_1\ f_1, \ldots)\ \{\textbf{super}(g_1, \ldots); \textbf{this}.f_1 = f_1; \ldots\}$
  - definiert den Konstruktor für Klasse C
  - vollständig bestimmt durch die Felder von C und der Superklassen
  - Anzahl der Parameter = Anzahl der Felder in C und den Superklassen
  - Rumpf beginnt mit $\textbf{super}(g_1, \ldots)$, wobei $g_1, \ldots$ den Feldern der Superklassen entsprechen
- $D\ m(C_1\ x_1, \ldots)\ \{\textbf{return}\ t;\}$
  - definiert Methode m
  - Ergebnistyp D
  - Parameter $x_1 \ldots$ mit Typen $C_1 \ldots$
  - Rumpf ist eine **return**-Anweisung

Die Klassentafel

- Die *Klassentafel CT* ist Abbildung von Klassennamen auf Klassendefinitionen
  - $\Rightarrow$ jede Klasse hat genau eine Definition
  - *CT* ist global verfügbar, entspricht dem Programm
  - "beliebig, aber fest"
- Jede Klasse außer Object hat eine Superklasse
  - Object taucht nicht in der Klassentafel auf
  - Object hat keine Felder
  - Object besitzt keine Methoden ($\neq$ Java)
- Die Klassentafel definiert eine Subtyprelation $C <: D$ auf Klassennamen
  - reflexive, transitive Hülle der Subklassendeklarationen

Die Subtyprelation

$$\text{REFL}\ \frac{}{C <: C}$$

$$\text{TRANS}\ \frac{C <: D \qquad D <: E}{C <: E}$$

$$\text{EXT}\ \frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D \ldots}{C <: D}$$

Konsistenz der Klassentafel

1. $CT(C) = \textbf{class}\ C \ldots$ für alle $C \in dom(CT)$
2. Object $\notin dom(CT)$
3. Für jeden Klassenname C, der in CT erscheint, gilt $C \in dom(CT) \cup \{\text{Object}\}$
4. Die Relation <: ist antisymmetrisch (keine Zyklen)

Beispiel: Klassen dürfen sich gegenseitig referenzieren

```
class Author extends Object {
  String name; Book bk;

  Author (String name, Book bk) {
    super();
    this.name = name;
    this.bk = bk;
  }
}
class Book extends Object {
  String title; Author ath;

  Book (String title, Author ath) {
    super();
    this.title = title;
    this.ath = ath;
  }
}
```

Hilfsdefinitionen

Felder zu einer Klasse bestimmen

$$\mathit{fields}(\text{Object}) = \bullet$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \, \{ C_1 \; f_1; \ldots \; K \; M_1 \ldots \} \qquad \mathit{fields}(D) = D_1 \; g_1, \ldots}{\mathit{fields}(C) = D_1 \; g_1, \ldots, C_1 \; f_1, \ldots}$$

- $\bullet$ — leere Liste
- $\mathit{fields}(\text{Author}) = \text{String name; Book bk;}$
- Verwendung: Berechnungsschritt, Typregeln

Hilfsdefinitionen

Typ einer Methode bestimmen

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \quad M_j = E\ m(E_1\ x_1, \ldots)\ \{\textbf{return } t;\}}{mtype(m, C) = (E_1, \ldots) \to E}$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \quad (\forall j)\ M_j \neq F\ m(F_1\ x_1, \ldots)\ \{\textbf{return } t;\} \quad mtype(m, D) = (E_1, \ldots) \to E}{mtype(m, C) = (E_1, \ldots) \to E}$$

- Verwendung: Typregeln

Hilfsdefinitionen

Rumpf einer Methode bestimmen

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \quad M_j = E\ m(E_1\ x_1, \ldots)\ \{\textbf{return } t;\}}{mbody(m, C) = (x_1 \ldots, t)}$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \quad (\forall j)\ M_j \neq F\ m(F_1\ x_1, \ldots)\ \{\textbf{return } t;\} \quad mbody(m, D) = (y_1 \ldots, u)}{mbody(m, C) = (y_1 \ldots, u)}$$

- Verwendung: Berechnungsschritt

Hilfsdefinitionen

Korrektes Überschreiben einer Methode

$$override(m, \text{Object}, (E_1 \dots) \to E)$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \{C_1\ f_1; \dots\ K\ M_1 \dots\} \quad M_j = E\ m(E_1\ x_1, \dots) \{\textbf{return } t;\}}{override(m, C, (E_1 \dots) \to E)}$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \{C_1\ f_1; \dots\ K\ M_1 \dots\} \quad (\forall j)\ M_j \neq F\ m(F_1\ x_1, \dots) \{\textbf{return } t;\} \quad override(m, D, (E_1, \dots) \to E)}{override(m, C, (E_1, \dots) \to E)}$$

- Verwendung: Typregeln

Beispiel

```
class Recording extends Object {
    int high; int today; int low;
    Recording (int high, int today, int low) { ... }
    int dHigh() { return this.high; }
    int dLow() { return this.low }
    String unit() { return "not set"; }
    String asString() {
        return String.valueOf(high)
            .concat("–")
            .concat (String.valueOf(low))
            .concat (unit());
    }
}
class Temperature extends ARecording {
  Temperature (int high, int today, int low) { super(high, today, low); }
  String unit() { return "°C"; }
}
```

- *fields*(Object) = •
- *fields*(Temperature) = *fields*(Recording) = int high; int today; int low;
- *mtype*(unit, Recording) = () → String
- *mtype*(unit, Temperature) = () → String
- *mtype*(dHigh, Recording) = () → int
- *mtype*(dHigh, Temperature) = () → int
- *override*(dHigh, Object, () → int)
- *override*(dHigh, Recording, () → int)
- *override*(dHigh, Temperature, () → int)
- *mbody*(unit, Recording) = (ε,"not set")
- *mtype*(unit, Temperature) = (ε,"°C")

Operationelle Semantik (Definition der Berechnungsschritte)

Direkte Rechenschritte

- Auswertung: Relation $t \longrightarrow t'$ für einen Berechnungsschritt

$$\text{E-ProjNew}\ \frac{\mathit{fields}(C) = C_1\ f_1, \ldots}{(\textbf{new}\ C(v_1, \ldots)).f_i \longrightarrow v_i}$$

$$\text{E-InvkNew}\ \frac{\mathit{mbody}(m, C) = (x_1 \ldots, t)}{\begin{array}{l}(\textbf{new}\ C(v_1, \ldots)).m(u_1, \ldots) \\ \longrightarrow t[\textbf{new}\ C(v_1, \ldots)/\mathsf{this}, u_1, \ldots/x_1, \ldots]\end{array}}$$

$$\text{E-CastNew}\ \frac{C <: D}{(D)(\textbf{new}\ C(v_1, \ldots)) \longrightarrow \textbf{new}\ C(v_1, \ldots)}$$

Rechenschritte im Kontext

$$\textsc{E-Field}\ \frac{t \longrightarrow t'}{t.f \longrightarrow t'.f}$$

$$\textsc{E-Invk-Recv}\ \frac{t \longrightarrow t'}{t.m(t_1, \ldots) \longrightarrow t'.m(t_1, \ldots)}$$

$$\textsc{E-Invk-Arg}\ \frac{t_i \longrightarrow t_i'}{v.m(v_1, \ldots, t_i, \ldots) \longrightarrow v.m(v_1, \ldots, t_i', \ldots)}$$

$$\textsc{E-New-Arg}\ \frac{t_i \longrightarrow t_i'}{\textbf{new}\ C(v_1, \ldots, t_i, \ldots) \longrightarrow \textbf{new}\ C(v_1, \ldots, t_i', \ldots)}$$

$$\textsc{E-Cast}\ \frac{t \longrightarrow t'}{(C)t \longrightarrow (C)t'}$$

Beispiel: Rechenschritte

((Pair) (**new** Pair (**new** Pair (**new** A(), **new** B()).setfst (**new** B()), **new** B()).fst)).fst

// → *[E−Field]*, *[E−Cast]*, *[E−New−Arg]*, *[E−InvkNew]*

((Pair) (**new** Pair (**new** Pair (**new** B(), **new** B()), **new** B()).fst)).fst

// → *[E−Field]*, *[E−Cast]*, *[E−ProjNew]*

((Pair) (**new** Pair (**new** B(), **new** B()))).fst

// → *[E−Field]*, *[E−CastNew]*

(**new** Pair (**new** B(), **new** B())).fst

// → *[E−ProjNew]*

new B()

Typregeln

Typregeln

Beteiligte Urteile

- $C <: D$
  C ist Subtyp von D
- $A \vdash t : C$
  unter Typannahme A hat Ausdruck t den Typ C
- $F\ m(C_1\ x_1, \ldots)\ \{\textbf{return}\ t;\}$ OK in C
  Methodendeklaration ist akzeptabel in Klasse C
- **class** C **extends** $D\ \{C_1\ f_1; \ldots\ K\ M_1 \ldots\}$ OK
  Klassendeklaration ist akzeptabel
- mit

$$A ::= \emptyset \mid A, x : C$$

Akzeptable Klassendeklarationen

$$\frac{\begin{array}{c} K = C(D_1\ g_1, \ldots, C_1\ f_1, \ldots)\ \{\textbf{super}(g_1, \ldots); \textbf{this}.f_1 = f_1; \ldots\} \\ \mathit{fields}(D) = D_1\ g_1 \ldots \\ (\forall j)\ M_j \text{ OK in } C \end{array}}{\textbf{class}\ C\ \textbf{extends}\ D\ \{C_1\ f_1; \ldots\ K\ M_1 \ldots\}}$$

Akzeptable Methodendeklarationen

$$\frac{\begin{array}{c} x_1 : C_1, \ldots, \mathsf{this} : C \vdash t : E \\ E <: F \\ CT(C) = \textbf{class } C \textbf{ extends } D \ldots \\ \mathit{override}(m, D, (C_1, \ldots) \to F) \end{array}}{F\ m(C_1\ x_1, \ldots)\ \{\textbf{return } t;\}\ \mathsf{OK\ in}\ C}$$

Akzeptable Terme haben einen Typ

$$\text{T-Var}\ \frac{x : C \in A}{A \vdash x : C}$$

$$\text{T-Field}\ \frac{A \vdash t : C \qquad \mathit{fields}(C) = C_1\ f_1, \ldots}{A \vdash t.f_i : C_i}$$

$$\text{F-Invk}\ \frac{\begin{array}{c} A \vdash t : C \qquad (\forall i)\ A \vdash t_i : C_i \qquad (\forall i)\ C_i <: D_i \\ \mathit{mtype}(m, C) = (D_1, \ldots) \to D \end{array}}{A \vdash t.m(t_1, \ldots) : D}$$

$$\text{F-New}\ \frac{\begin{array}{c} (\forall i)\ A \vdash t_i : C_i \qquad (\forall i)\ C_i <: D_i \\ \mathit{fields}(C) = D_1\ f_1, \ldots \end{array}}{A \vdash \textbf{new}\ C(t_1, \ldots) : C}$$

Typregeln für Typcast

$$\text{T-UCast}\ \frac{A \vdash t : D \qquad D <: C}{A \vdash (C)t : C}$$

$$\text{T-DCast}\ \frac{A \vdash t : D \qquad C <: D \qquad C \neq D}{A \vdash (C)t : C}$$

Typsicherheit für Featherweight Java

- Typsicherheit folgt aus "Preservation" und "Progress"
- "Preservation":
  Falls $A \vdash t : C$ und $t \longrightarrow t'$, dann ist $A \vdash t' : C'$ für $C' <: C$.
- "Progress": (abgekürzt)
  Falls $A \vdash t : C$, dann ist entweder $t \equiv v$ ein Wert oder t enthält einen Subterm der Form

  $$(C)(\textbf{new}\ D(v_1, \ldots))$$

  wobei $D \not<: C$.
- Also:
  - Alle Methodenaufrufe und Feldzugriffe laufen fehlerfrei ab
  - Typcasts können fehlschlagen

Problem aus dem Preservation Beweis

Typcasts zerstören Preservation

- Betrachte den Term (A) ((Object)**new** B())
- Es gilt $\emptyset \vdash$(A) ((Object)**new** B()): A
- Es gilt (A) ((Object)**new** B()) $\longrightarrow$ (A) (**new** B())
- Aber (A) (**new** B()) hat keinen Typ!

Problem aus dem Preservation Beweis

Typcasts zerstören Preservation

- Betrachte den Term (A) ((Object)**new** B())
- Es gilt $\emptyset \vdash$(A) ((Object)**new** B()): A
- Es gilt (A) ((Object)**new** B()) $\longrightarrow$ (A) (**new** B())
- Aber (A) (**new** B()) hat keinen Typ!
- Abhilfe: zusätzliche Regel *stupid cast* für diesen Fall —nächster Berechnungsschritt schlägt fehl!

$$\text{T-SCast}\ \frac{A \vdash t : D \qquad C \not<: D \qquad D \not<: C}{A \vdash (C)t : C}$$

- Mit dieser Regel lässt sich Preservation beweisen

Aussage der Typsicherheit

Wenn $A \vdash t : C$, dann liegt einer der folgenden Fälle vor

1. t terminiert nicht,
   d.h. es gibt eine unendliche Folge von Berechnungsschritten

$$t = t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$$

2. t liefert nach endlich vielen Schritten einen Wert v,
   d.h. es gibt eine endliche Folge von Berechnungsschritten

$$t = t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n = v$$

3. t enthält nach endlich vielen Schritten einen Subterm der Form

$$(C)(\textbf{new}\ D(v_1, \ldots))$$

   wobei $D \not<: C$.