Objektorientierung
Schritt für Schritt

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn
Email: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf`

Januar 2007

(Die Folien sind online verfügbar: `.../~ralf/talks.html#T53`.)

Einleitung

Um das Thema Objektorientierung rankt sich ein üppiger Begriffsdschungel:

Klasse, anonyme Klasse, Objekt, Instanz, Feld, Instanzvariable, Methode, Schnittstelle, Sichtbarkeit, Konstruktor, Methodenaufruf, Klassenvariable, Klassenmethode, innere Klasse, Kapselung, Verhalten, dynamische Bindung, späte Bindung, offene Rekursion, Objekterschaffung, Vererbung, Mehrfachvererbung, Hierarchie, Oberklasse, Unterklasse, Redefinition, Überschreibung, Polymorphismus, Schnittstellenvererbung, Obertyp, Untertyp, generische Klasse, Methodentabelle

sind einige der Schlagworte, die in diesem Zusammenhang genannt werden.

☞ Ist Objektorientierung komplex?

Einleitung

You can never understand one language
until you understand at least two.
Ronald Searle (1920–)

Make everything as simple as possible,
but not simpler.
Albert Einstein (1859–1955)

- *Ziel:* eine Sprache evolutionär um oo Konzepte erweitern;
- in drei einfachen (?) Schritten;
- Syntax *und* Semantik präzise definieren.
- *Jeweils:* den Kern des Konzeptes herausarbeiten;
- das Konzept so einfach wie möglich gestalten;
- Konzepte orthogonal halten — in der Tradition von Algol.

Überblick

Endliche Abbildungen

Sind X und Y Mengen, dann bezeichnet $X \rightarrow_{\text{fin}} Y$ die Menge aller endlichen Abbildungen von X nach Y. Der Definitionsbereich einer endlichen Abbildung φ wird mit *dom* φ bezeichnet.

- *Einelementige Abbildung* (Notation: $\{x \mapsto y\}$):
  - $dom\{x \mapsto y\} = \{x\}$
  - $\{x \mapsto y\}(x) = y$
- *Erweiterung* von φ_1 um φ_2 (Notation: φ_1, φ_2):
  - $dom\ (\varphi_1, \varphi_2) = dom\ \varphi_1 \cup dom\ \varphi_2$
  - $(\varphi_1, \varphi_2)(x) = \begin{cases} \varphi_2(x) & \text{if } x \in dom\ \varphi_2 \\ \varphi_1(x) & \text{otherwise} \end{cases}$

Der funktionale Kern — Beispiel: binäre Suche

```
function binary-search (oracle : Nat → Bool,
                        lower-bound : Nat,
                        upper-bound : Nat) : Nat =
  let
    function search (l : Nat, u : Nat) : Nat =
      if l ⩾ u then u
               else let
                      val m = (l + u) ÷ 2
                    in
                      if oracle m then search (l,     m)
                                  else search (m + 1, u)
                    end
  in
    search (lower-bound, upper-bound)
  end
```

Boolesche Werte — abstrakte Syntax

$e \in$	Expr	*Ausdrücke*
$e ::=$	*false*	falsch
$\mid$	*true*	wahr
$\mid$	**if** e_1 **then** e_2 **else** e_3	Alternative

Boolesche Werte — statische Semantik

$\tau \in \text{Type}$	*Typen*
$\tau ::= \mathit{Bool}$	Typ der Booleschen Werte

Typregeln:

$$\frac{}{\mathit{false} : \mathit{Bool}} \qquad \frac{}{\mathit{true} : \mathit{Bool}}$$

$$\frac{e_1 : \mathit{Bool} \qquad e_2 : \tau \qquad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

Boolesche Werte — dynamische Semantik

$\nu \in \text{Val}$	*Werte*
$\nu ::= \mathit{false}$	falsch
$\mid \mathit{true}$	wahr

Auswertungsregeln:

$$\frac{}{\mathit{false} \Downarrow \mathit{false}} \qquad \frac{}{\mathit{true} \Downarrow \mathit{true}}$$

$$\frac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow \nu}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow \nu} \qquad \frac{e_1 \Downarrow \mathit{false} \qquad e_3 \Downarrow \nu}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow \nu}$$

Lokale Deklarationen — abstrakte Syntax

$x \in$	Id	*Bezeichner*
$e ::= \cdots$		
$\mid$	x	Bezeichner
$\mid$	**let** d **in** e **end**	lokale Deklaration

$d \in$	Decl	*Deklarationen*
$d ::=$	**val** $x = e$	Wertedefinition
$\mid$	$d_1\ d_2$	Sequenz von Deklarationen
$\mid$	**local** d_1 **in** d_2 **end**	lokale Deklaration

Lokale Deklarationen — statische Semantik

$\Sigma \in \mathsf{Sig} = \mathsf{Id} \to_{\mathrm{fin}} \mathsf{Type}$ *Signaturen*

$$\frac{}{\Sigma \vdash x : \Sigma(x)} \qquad x \in \mathit{dom}\ \Sigma$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \vdash e : \tau}{\Sigma \vdash \textbf{let}\ d\ \textbf{in}\ e\ \textbf{end} : \tau}$$

$$\frac{\Sigma \vdash e : \tau}{\Sigma \vdash (\textbf{val}\ x = e) : \{x \mapsto \tau\}}$$

$$\frac{\Sigma \vdash d_1 : \Sigma_1 \qquad \Sigma, \Sigma_1 \vdash d_2 : \Sigma_2}{\Sigma \vdash d_1\ d_2 : \Sigma_1, \Sigma_2}$$

$$\frac{\Sigma \vdash d_1 : \Sigma_1 \qquad \Sigma, \Sigma_1 \vdash d_2 : \Sigma_2}{\Sigma \vdash \textbf{local}\ d_1\ \textbf{in}\ d_2\ \textbf{end} : \Sigma_2}$$

☞ Spätere Definitionen verschatten frühere.

Lokale Deklarationen — dynamische Semantik

$\delta \in \mathsf{Env} = \mathsf{Id} \rightarrow_{\mathrm{fin}} \mathsf{Val}$ *Umgebungen*

$$\frac{d \Downarrow \delta \qquad e\delta \Downarrow \nu}{\textbf{let } d \textbf{ in } e \textbf{ end} \Downarrow \nu}$$

$$\frac{e \Downarrow \nu}{(\textbf{val } x = e) \Downarrow \{x \mapsto \nu\}}$$

$$\frac{d_1 \Downarrow \delta_1 \qquad d_2\delta_1 \Downarrow \delta_2}{d_1\ d_2 \Downarrow \delta_1, \delta_2}$$

$$\frac{d_1 \Downarrow \delta_1 \qquad d_2\delta_1 \Downarrow \delta_2}{\textbf{local } d_1 \textbf{ in } d_2 \textbf{ end} \Downarrow \delta_2}$$

☞ $e\delta$ ist die Anwendung einer *Substitution*: jede freie Variable x in e mit $x \in \mathit{dom}\ \delta$ wird durch $\delta(x)$ ersetzt.

Der imperative Kern — Beispiel: Bankkonkto

```
local
  val bal = ref 0
in
  function deposit (amount : Nat) : () =
    bal := !bal + amount
  function withdraw (amount : Nat) : () =
    bal := !bal − amount
  function balance () : Nat =
    ! bal
end
```

Speicher — abstrakte Syntax

$e ::= \cdots$		
$\mid$	**ref** e	Allokation
$\mid$	$!\, e$	Dereferenzierung
$\mid$	$e_1 := e_2$	Zuweisung

☞ Das Konzept des Speichers ist explizit.

Speicher — statische Semantik

$$\tau ::= \cdots \mid \mathit{Ref}\ \langle \tau \rangle \qquad \text{Referenztyp}$$

$$\frac{\Sigma \vdash e : \tau}{\Sigma \vdash \mathbf{ref}\ e : \mathit{Ref}\ \langle \tau \rangle}$$

$$\frac{\Sigma \vdash e : \mathit{Ref}\ \langle \tau \rangle}{\Sigma \vdash\ !e : \tau}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Ref}\ \langle \tau \rangle \qquad \Sigma \vdash e_2 : \tau}{\Sigma \vdash e_1 := e_2 : ()}$$

Speicher — dynamische Semantik

$a \in \text{Addr}$		*Adressen*
$\nu ::= \cdots$		
	a	Adresse
$\sigma \in \text{Addr} \to_{\text{fin}} \text{Val}$		*Speicher*

$$\frac{\sigma \mid e \Downarrow \nu \mid \sigma'}{\sigma \mid \textbf{ref}\ e \Downarrow a \mid \sigma', \{a \mapsto \nu\}} \qquad a \notin \mathit{dom}\ \sigma'$$

$$\frac{\sigma \mid e \Downarrow a \mid \sigma'}{\sigma \mid !e \Downarrow \sigma'(a) \mid \sigma'}$$

$$\frac{\sigma \mid e_1 \Downarrow a \mid \sigma_1 \qquad \sigma_1 \mid e_2 \Downarrow \nu \mid \sigma_2}{\sigma \mid e_1 := e_2 \Downarrow () \mid \sigma_2, \{a \mapsto \nu\}}$$

Kapselung — Beispiel: Bankkonto

```
val bank-account : Bank-Account =
  class
    local
      val bal = ref 0
    in
      method deposit (amount : Nat) : () =
        bal := !bal + amount
      method withdraw (amount : Nat) : () =
        bal := !bal − amount
      method balance : Nat =
        ! bal
    end
  end
val my-account = new bank-account
```

☞ Eine *Klasse* ist durch einen Ausdruck gegeben.

Kapselung — Beispiel: Bankkonto

type *Bank-Account* =
  class
    method *deposit* : $Nat \to ()$
    method *withdraw* : $Nat \to ()$
    method *balance* : Nat
  end

☞ Eine *Schnittstelle* ist ein Typ, der Typ einer Klasse.

Kapselung — abstrakte Syntax

$e ::= \cdots$	
$\mid$ **class** m **end**	anonyme Klasse
$\mid$ **new** e	Objekterschaffung
$\mid$ $e.x$	Methodenaufruf

☞ Eine Klasse ist eine Sammlung von Methoden.

$m \in$ Method	*Methodendeklarationen*
$m ::=$ **method** $x : \tau = e$	Methodendefinition
$\mid$ $m_1\ m_2$	Sequenz von Deklarationen
$\mid$ **local** d **in** m **end**	lokale Deklaration

☞ Instanzvariablen (Felder in Java) sind nur lokal sichtbar (Kapselung).

Kapselung — statische Semantik

$M \in \mathsf{Id} \to_{\mathrm{fin}} \mathsf{Type}$ — *Methodensignaturen*

$\tau ::= \cdots$

$\mid$ **class** M **end**	Klassentyp
$\mid$ **object** M **end**	Objekttyp

☞ Ein Objekttyp entspricht einer Schnittstelle.

$$\frac{\Sigma \vdash m : M}{\Sigma \vdash \textbf{class}\ m\ \textbf{end} : \textbf{class}\ M\ \textbf{end}}$$

$$\frac{\Sigma \vdash e : \textbf{class}\ M\ \textbf{end}}{\Sigma \vdash \textbf{new}\ e : \textbf{object}\ M\ \textbf{end}}$$

$$\frac{\Sigma \vdash e : \textbf{object}\ M\ \textbf{end}}{\Sigma \vdash e.x : M(x)} \qquad x \in \mathit{dom}\ M$$

Kapselung — statische Semantik

$$\frac{\Sigma \vdash e : \tau}{\Sigma \vdash (\textbf{method}\ x : \tau = e) : \{x \mapsto \tau\}}$$

$$\frac{\Sigma \vdash m_1 : M_1 \qquad \Sigma \vdash m_2 : M_2}{\Sigma \vdash m_1\ m_2 : M_1, M_2}$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \vdash m : M}{\Sigma \vdash \textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} : M}$$

☞ Spätere Definitionen verschatten frühere (Redefinition).

Kapselung — Introduktion und Elimination

Konstrukte einer Programmiersprache können in Introduktions- und Eliminationsformen aufgeteilt werden.

class *m* **end**	Introduktion
class *M* **end**	
new *e*	Elimination & Introduktion
object *M* **end**	
e.*x*	Elimination

☞ **new** eliminiert eine Klasse und führt ein Objekt ein.

Kapselung — dynamische Semantik

$\mu \in \mathrm{Id} \to_{\mathrm{fin}} \mathrm{Expr}$ — *Methodenumgebungen*

$\nu ::= \cdots$

$\mid$ **class** m **end**	anonyme Klasse
$\mid$ **object** μ **end**	Objekt

☞ μ ist eine Methodentabelle (dispatch table).

$$\frac{}{\textbf{class } m \textbf{ end} \Downarrow \textbf{class } m \textbf{ end}}$$

$$\frac{e \Downarrow \textbf{class } m \textbf{ end} \qquad m \Downarrow \mu}{\textbf{new } e \Downarrow \textbf{object } \mu \textbf{ end}}$$

$$\frac{e \Downarrow \textbf{object } \mu \textbf{ end} \qquad \mu(x) \Downarrow \nu}{e.x \Downarrow \nu}$$

☞ Die Auswertung einer Klasse wird verzögert; **new** forciert die Auswertung; die Auswertung der Methoden wird verzögert (Folie 32); Methodenaufrufe forcieren die Auswertung (dynamische Bindung).

Kapselung — dynamische Semantik

$$\frac{}{(\textbf{method}\ x : \tau = e) \Downarrow \{x \mapsto e\}}$$

$$\frac{m_1 \Downarrow \mu_1 \qquad m_2 \Downarrow \mu_2}{m_1\ m_2 \Downarrow \mu_1, \mu_2}$$

$$\frac{d \Downarrow \delta \qquad m\delta \Downarrow \mu}{\textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} \Downarrow \mu}$$

☞ Methoden werden nicht ausgewertet. Warum?

- **method** *balance* : *Nat* ist *keine* natürliche Zahl, sondern eine Berechnung, die zu einer natürlichen Zahl auswertet.
- *Später:* der Methodenrumpf kann den speziellen Bezeichner *self* enthalten, der für das Objekt selbst steht und der erst beim Methodenaufruf gebunden wird (late binding).

Kapselung — Lebenszyklen von Klassen/Objekten

val *bank-account* : *Bank-Account* =
  class local val *bal* = **ref** 0
    in **method** *deposit* (*amount* : *Nat*) : () =
      bal := !*bal* + *amount*
  . . . **end end**

☞ *bank-account* wird an den Klassenausdruck gebunden; die Auswertung von **class** . . . **end** wird verzögert.

val *my-account* = **new** *bank-account*

☞ *my-account* wird an eine Instanz von *bank-account* gebunden; **new** forciert die Auswertung der Klasse: eine Speicherzelle wird allokiert. Die Auswertung der Methode *deposit* wird verzögert.

my-account.deposit (4711)

☞ Die Methode *deposit* wird ausgewertet: 4711 wird zu dem Inhalt von *bal* addiert.

Kapselung — Konstruktoren

```
function bank-account (money : Nat) : Bank-Account =
  class
    local
      val bal = ref money
    in
      method deposit (amount : Nat) : () =
        bal := !bal + amount
      method withdraw (amount : Nat) : () =
        bal := !bal − amount
      method balance : Nat =
        ! bal
    end
  end
val my-account = new (bank-account 4711)
```

☞ Ein Konstruktor ist ein Ausdruck, der zu einer Klasse auswertet — in der Regel eine Funktion.

Kapselung — Klassenvariablen und -methoden

```
local
  val no = ref 0
in
  function number-of-accounts () : Nat = !no
  val bank-account =
    class
      local
        val bal = no := !no + 1; ref 0
      in
        method deposit (amount : Nat) : () = ...
        ...
      end
    end
end
```

☞ Eine Klassenvariable wird außerhalb einer Klasse definiert; eine Klassenmethode ist einfach eine Funktion.

Kapselung — klassenerzeugende Klassen

```
val bank =
  class
    local val no = ref 0
    in
      method number-of-accounts : Nat = !no
      method account : Bank-Account =
        class
          local
            val bal = no := !no + 1; ref 0
          in
            method deposit (amount : Nat) : () = ...
            ....
          end
        end
    end
  end
```

☞ Vorher: ein Bankkonto $\rightsquigarrow$ viele Bankkonten; hier: eine Bank $\rightsquigarrow$ viele Banken.

Untertypen — statische Semantik

Beobachtung: $e.x$ ist zulässig, sofern e mindestens über eine x Methode verfügt.

$$\frac{\Sigma \vdash e : \tau \qquad \tau \preccurlyeq \tau'}{\Sigma \vdash e : \tau'}$$

$$\frac{M(x) \preccurlyeq M'(x) \quad | \quad x \in dom\ M}{\textbf{class}\ M\ \textbf{end} \preccurlyeq \textbf{class}\ M'\ \textbf{end}} \qquad dom\ M' \subseteq dom\ M$$

$$\frac{M(x) \preccurlyeq M'(x) \quad | \quad x \in dom\ M}{\textbf{object}\ M\ \textbf{end} \preccurlyeq \textbf{object}\ M'\ \textbf{end}} \qquad dom\ M' \subseteq dom\ M$$

☞ Inklusionstyppolymorphismus (Schnittstellenvererbung).

☞ Untertypen erleichtern die *Verwendung* einer Klasse.

Vererbung — Beispiel: erweitertes Bankkonto

```
val extended-bank-account =
   class
      inherit bank-account
      method clear : Nat =
         let
            val money = self.balance
         in
            self.withdraw money;
            money
         end
   end
```

☞ Nur Methoden werden vererbt (Vererbung).

☞ Mehrfachvererbung mittels mehrerer **inherit** Deklarationen.

☞ Vererbung erleichtert die *Implementierung* einer Klasse.

Vererbung — abstrakte Syntax

$m ::= \cdots$		*method declaration*
	inherit e	inherit from a class

☞ *Zusätzlich:* Methodenrümpfe dürfen den speziellen Bezeichner *self* (*this* in Java) enthalten.

☞ *self* ermöglicht Rekursion (offene Rekursion).

Vererbung — statische Semantik

$$\frac{\Sigma \mid \textbf{object}\ M\ \textbf{end} \vdash m : M}{\Sigma \vdash \textbf{class}\ m\ \textbf{end} : \textbf{class}\ M\ \textbf{end}}$$

$$\frac{\Sigma, \{\mathit{self} \mapsto \gamma\} \vdash e : \tau}{\Sigma \mid \gamma \vdash (\textbf{method}\ x : \tau = e) : \{x \mapsto \tau\}}$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \mid \gamma \vdash m : M}{\Sigma \mid \gamma \vdash \textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} : M}$$

$$\frac{\Sigma \vdash e : \textbf{class}\ M\ \textbf{end}}{\Sigma \mid \gamma \vdash \textbf{inherit}\ e : M}$$

☞ *self* ist nur in den Methodenrümpfen sichtbar. Warum?

- *self* wird erst gebunden, wenn eine Methode aufgerufen wird (späte Bindung).

Vererbung — dynamische Semantik

$$\frac{e \Downarrow \textbf{object}\ \mu\ \textbf{end} \qquad \mu(x)\{\mathit{self} \mapsto \textbf{object}\ \mu\ \textbf{end}\} \Downarrow \nu}{e.x \Downarrow \nu}$$

$$\frac{e \Downarrow \textbf{class}\ m\ \textbf{end} \qquad m \Downarrow \mu}{\textbf{inherit}\ e \Downarrow \mu}$$

☞ *self* wird an das Objekt selbst gebunden.

☞ **inherit** ähnelt **new**.

☞ Sichtbarkeit: kein Unterschied zwischen **new** und **inherit**. Das ist eine bewusste Designentscheidung.

Zusammenfassung

- Objektorientierung in drei einfachen (?) Schritten.
- Die resultierende Sprache ist ausdrucksstark.
- Klassen sind Werte (first-class values).
- Untertypen (Polymorphismus) versus Unterklassen (Vererbung).

Ausblick:

- *Singleton pattern*, Klassen mit genau einer Instanz: **new class** . . . **end**.
- Generische Klassen mittels polymorpher Konstruktorfunktionen — keine Erweiterung nötig.

super — Beispiel: Bankkonto mit Pegelstand

```
val max-bank-account =
  class
    inherit
      super = extended-bank-account
    in
      local
        val max = ref (super.balance)
      in
        method deposit (amount : Nat) : () =
          super.deposit amount;
          max := maximum (!max, super.balance)
        method max-balance : Nat =
          ! max
      end
    end
  end
```

super — abstrakte Syntax

$m ::= \cdots$
$\quad \mid$ **inherit** $x = e$ **in** m **end** Vererbung

☞ x ist der Name des Oberobjekts; e ist die Oberklasse.

☞ Überschreibung (overriding) ist einfach: einfach eine neue Methode mit dem gleichen Namen definieren.

super — statische Semantik

$$\frac{\Sigma \vdash e : \textbf{class } M_1 \textbf{ end} \qquad \Sigma, \{x \mapsto \textbf{object } M_1 \textbf{ end}\} \mid \gamma \vdash m : M_2}{\Sigma \mid \gamma \vdash \textbf{inherit } x = e \textbf{ in } m \textbf{ end} : M_1, M_2}$$

super — dynamische Semantik

$$\begin{array}{lll} \nu ::= \cdots & & \\ \quad \mid \textbf{super } \mu \textbf{ of } \nu \textbf{ end} & & \text{Oberobjekt und '}\textit{self}\text{'} \end{array}$$

$$\frac{e \Downarrow \textbf{class } m_1 \textbf{ end} \qquad m_1 \Downarrow \mu_1 \qquad m\{x \mapsto \textbf{super } \mu_1 \textbf{ of } \textit{self} \textbf{ end}\} \Downarrow \mu_2}{\textbf{inherit } x = e \textbf{ in } m \textbf{ end} \Downarrow \mu_1, \mu_2}$$

$$\frac{e \Downarrow \textbf{super } \mu \textbf{ of } o \textbf{ end} \qquad \mu(x)\{\textit{self} \mapsto o\} \Downarrow \nu}{e.x \Downarrow \nu}$$

☞ Trickreich!

- x wird an das Oberobjekt gebunden;
- die zweite Komponente von **super** μ **of** ν **end** wird auf *self* gesetzt;
- wenn eine Methode des Unterobjekts aufgerufen wird, dann wird *self* gebunden;
- dieses Objekt wird an die Methoden der Oberklasse weitergereicht (super call).