Wiederverwendung

Komponenten in mehr als einem Kontext verwenden:

klassisch: Modularisierung, Parametrisierung

objektorientiert:

- Objekt durch anderes Objekt ersetzen (inheritance)
- Methode durch andere Methode ersetzen (overriding)

Methoden greifen auf *selbst* zu. ABER: *wer bin ich?*

Objekt-Orientierung

Software modelliert reale (z. B. physikalische) Systeme

Bestandteile des Systems ⇒ Bestandteile des Modells

reale Objekte ⇒ Software-Objekte

Smalltalk (ca. 1980): *everything is an object*

robuste Modelle, nachnutzbare Modell-Bestandteile

Martin Abadi und Luca Cardelli:

A Theory of Objects, Springer 1996

`http://www.luca.demon.co.uk/`

`TheoryOfObjects.html`

Johannes Waldmann
Institut für Informatik, Universität Leipzig

Objekttypen erster Ordnung

Typen $A, B, \ldots$ sind $[l_1 : B_1, \ldots]$ und Grundtyp K

Terme $a, b, \ldots$ sind $[l_1 = \varsigma(x_1 : A_1)b_1, \ldots]$, $\quad a.l, \quad a.l := \varsigma(x : A)b$

Typisierungs-Regeln

Wenn $A = [l_1 : B_1, \ldots]$ dann $\dfrac{E, x_1 : A_1 \vdash b_1 : B_1, \ldots}{E \vdash [l_1 = \varsigma(x_1 : A)b_1, \ldots] : A}$

$$\frac{E \vdash a : [l_1 : B_1, \ldots]}{E \vdash a.l_j : B_j} \qquad \frac{E \vdash a : A \quad E, x : A \vdash b : B_j}{E \vdash a.l_j := \varsigma(x : A)b : A}$$

Typ ist eindeutig, Subject Reduction, keine Normalisierung

Funktionen als Objekte

$\lceil \cdot \rceil$: Lambda-Kalkül $\to$ Objekt-Kalkül

- $\lceil fa \rceil = (\lceil f \rceil.\text{arg} := \varsigma(y)\lceil a \rceil).\text{val}$
- $\lceil \lambda(x)b \rceil = \begin{bmatrix} \text{arg} = \varsigma(x)x.\text{arg}, \\ \text{val} = \varsigma(x)b[x \leftarrow x.\text{arg}] \end{bmatrix}$

„kostenloser“ Lambda-Kalkül, verkürzt Notation.

Objekte als Lambda-Terme?

ungetypt: JA, getypt: NEIN. (Subtyp-Relationen verletzt)

Was ist ein Objekt?

- $\varsigma(x)b$: Methode mit self-Parameter x und Körper b
- $[l_1 = \varsigma(x_1)b_1, \ldots]$ Objekt mit benannten Methoden
- $o.l$ Methode l des Objekts o
- $o.l := \varsigma(x)b$ (Kopie von) o mit geänderter Methode l
  falls x nicht in b, dann kürzer $o.l := b$
  Kopieren: Vererbung (inheritance), Ändern: (overriding)

Objekt-Semantik $\to$

- $o.l_k \;\to\; b_k[x_k \leftarrow o]$ Methode auf *self* anwenden
- $o.l := m \;\to\; [\ldots, l = m, \ldots]$ Methode ändern

Wo ist der *self*-Typ?

Ziel: $P_2 <: P_1$ für

$$\begin{aligned} P_1 &\equiv [x:\mathbb{Z},\ \mathsf{move}_x : \mathbb{Z} \to P_1] \\ P_2 &\equiv [x, y:\mathbb{Z},\ \mathsf{move}_x, \mathsf{move}_y : \mathbb{Z} \to P_2] \end{aligned}$$

(bis jetzt verboten wegen Invarianz.) — Versuch:

$$\begin{aligned} P_1 &\equiv [x:\mathbb{Z},\ \mathsf{move}_x : \mathbb{Z} \to \mathsf{Self}] \\ P_2 &\equiv [x, y:\mathbb{Z},\ \mathsf{move}_x, \mathsf{move}_y : \mathbb{Z} \to \mathsf{Self}] \end{aligned}$$

$p : P_2 <: P_1$, betrachte $p' \equiv p.\mathsf{move}_x := \varsigma(y : P_1)\lambda(d : \mathbb{Z})c$

mit Konstante $c \equiv [x = 0, \mathsf{move}_x = \lambda(d:\mathbb{Z})(x + d)] : P_1$

$p'.\mathsf{move}_x(5) : P_2$, aber $p'.\mathsf{move}_x(5).y$??

Objekte mit kovarianten Komponenten?

Ansatz: $\dfrac{E \vdash B_1 <: B_1', \ldots}{E \vdash [l_1 : B_1, \ldots] <: [l_1 : B_1', \ldots]}$?

Beispiel: Typen $\mathbb{R}_+ <: \mathbb{R}$, Funktion $\log : \mathbb{R}_+ \to \mathbb{R}$.

für $P \equiv [x : \mathbb{R}, f : \mathbb{R}]$ gilt *nicht* $[x = 1.0, f = \varsigma(s : P)\log(s.x)] : P$.

$Q \equiv [x : \mathbb{R}_+, f : \mathbb{R}] <: P$ nach Ansatz

$a \equiv [x = 1.0, f = \varsigma(s : Q)\log(s.x)] : Q$, also auch $a : P$

$b \equiv a.x := -1.0$ wg $a : P$ gilt $b : P$, also $b.f : \mathbb{R}$

damit Widerspruch $b.f \to^* \log(-1.0) : \mathbb{R}$.

Subtyping

reflexive transitive Relation $<:$ auf Typen, $a : A \wedge A <: B \vdash a : B$

für *Paare*:

$$\frac{E \vdash A' <: A \quad E \vdash B' <: B}{E \vdash A' \times B' <: A \times B}$$

$T \times \cdot$ und $\cdot \times T$ sind ko-variant.

für *Funktionen*:

$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash (A \to B) <: (A' \to B')}$$

$T \to \cdot$ ist ko-variant,
$\cdot \to T$ ist kontra-variant.

für *Objekte*: $E \vdash [l_1 : B_1, \ldots, l_m : B_m, \ldots] <: [l_1 : B_1, \ldots, l_m : B_m]$

(in-variant!)

Eigenschaften: eindeutige Minimale Typen, Subject Reduction.

Fixpunkte für Typen

$\mu(X)B$: der Fixpunkt von $\lambda(X)B$ $\quad (X$ erscheint in $B)$

Beispiel: $B = 1 + (\mathbb{N} \times X)$, $\mu(X)B =$ Listen von Zahlen

Der Self-Quantor

$\varsigma(X)B \equiv \mu(Y)\exists(X <: Y)B \quad (X$ kovariant in B, Y nicht in $B)$

$$P_1 \equiv \varsigma(\mathsf{Self})[x : \mathbb{Z},\ \mathsf{move}_x : \mathbb{Z} \to \mathsf{Self}]$$
$$P_2 \equiv \varsigma(\mathsf{Self})[x, y : \mathbb{Z},\ \mathsf{move}_x, \mathsf{move}_y : \mathbb{Z} \to \mathsf{Self}] \ \boxed{<: P_1}$$

Existenziell quantifizierte Typen

modellieren (partiell) abstrakte Datentypen:

Typ $\exists(X <: A)B$ mit Interface B, versteckter Repräsentation X

```
pack X <: A = A' with b : B   -- X in B
open c as X <: A, x:B in d:D  -- x, X in d, aber X nicht in D
```

$$\frac{E \vdash A <: A' \quad E, X : A \vdash B <: B'}{E \vdash \exists(X <: A)B <: \exists(X <: A')B'}$$

kovariant in Schranke (A) und Körper (B)

```
data E = E (forall x . Show x => x)
size :: E -> Integer
size (E x) = length (show x)
```


Typen zweiter Stufe: parametrische Polymorphie

durch universell quantifizierte Typen:

$\mathsf{id} : \forall(X)X \to X \equiv \lambda(X)\lambda(x : X)x$, $\mathsf{id}(\mathbb{N}) : \mathbb{N} \to \mathbb{N}$, $\mathsf{id}(\mathbb{N})(3) : \mathbb{N}$.

beschränkte parametrische Polymorphie: $\forall(X <: A)B$

$$\frac{E \vdash A' <: A \quad E, X : A' \vdash B <: B'}{E \vdash \forall(X <: A)B <: \forall(X <: A')B'}$$

kontravariant in der Schranke (A), kovariant im Körper (B)

$$\frac{E, X : A \vdash b : B}{E \vdash \lambda(X <: A)b : \forall(X <: A)B} \qquad \frac{E \vdash b : \forall(X <: A)B \quad E \vdash A' <: A}{E \vdash b(A') : B[X \leftarrow A']}$$

Objekte, Updates, . . . Prozesse?

(reale) Objekte nehmen an stetigen Prozessen teil

Software-Objekte: Prozess ist diskrete Folge von Updates

Updates liefern schon genügend Schwierigkeiten für Typing

Objekte in Deklarativer Programmierung?

Code-Re-use durch Polymorphie, ADTs, Modularisierung

Klassen (Prädikate) sind einfacher und passen besser
(zum restlichen Typsystem) als primitive Objekt-Typen

mehrstellige Methoden, mehrparametrige Klassen (Prädikate)

(Jetzt nicht mehr Abadi/Cardelli, sondern eigene Mutmaßungen.)

Rein Klassen-basierte OO (Haskell, Mercury)

Trennung: Implementierung (Daten) von Interface (Methoden)

```
data Tree a = Node { key :: a, children :: [ Tree a ] }

class Size b where size :: b -> Integer
size :: forall b . Size b => b -> Integer

instance Size (Tree a) where
    size t = 1 + sum ( map size $ children t )
instance Size [a]      where    size = length
```

wichtig: `Size` ist kein *Typ*, sondern eine *Klasse*,
d. h. ein Prädikat: Typ → Bool.

Objekte und binäre Methoden

$P \equiv \varsigma(\mathsf{Self})[x : \mathbb{Z},\ \mathsf{move}_x : \mathbb{Z} \to \mathsf{Self}, \boxed{\mathsf{equal} : \mathsf{Self} \to \mathsf{Bool}}]$

kontravariantes Self — Ansatz ist ungeeignet. Ausweg:
unterscheide zwischen subtyping (<:) und *matching* (<#)

$$Q' \equiv \mu(X)[l : X \to X, m : X] \quad \begin{array}{c}\not<:\\ <\#\end{array} \quad Q \equiv \mu(X)[l : X \to X]$$

Vorsicht: $a : A \wedge A <\# B \not\vdash a : B$. Beispiel: $(f : Q \to C)(x : Q')$.

Benutze $\forall(Y <\# P)Y$ (Typklasse), OK für binäre Methoden

Objektorientierte Sprachen

- Ausführung: wie schnell läuft das Programm?
  Methoden-Lookup dauert, Inlining ist erschwert.
- Kompilation: wie schnell ist der Compiler?
  Superklassen rekompilieren? Kompilezeit > linear.
- Entwicklung im kleinen: wie arbeitet *ein* Programmierer?
  zunächst ist OO sehr effektiv,
- Entwicklung im großen: wie arbeitet ein *Team*?
  Typsysteme nicht ausdrucksstark genug, um Abstraktionen zu deklarieren und zu kontrollieren
- Konzepte: wie schwer ist die Sprache zu erlernen?
  viele interagierende Eigenschaften (Polymorphie, Subtyping, . . .)

Prozedurale Sprachen

- Ausführung: wie schnell läuft das Programm?
  Sehr schnell. Ohne Laufzeit-Typchecks: noch schneller.
- Kompilation: wie schnell ist der Compiler?
  Module, Interfaces, getrennte Kompilierung.
- Entwicklung im kleinen: wie arbeitet *ein* Programmierer?
  viele Entwurfs- und Tipp-Fehler durch Typchecking gefunden
- Entwicklung im großen: wie arbeitet ein *Team*?
  Daten-Abstraktion, Modularisierung
- Konzepte: wie schwer ist die Sprache zu erlernen?
  orthogonale Bausteine

Bad Engineering . . .

Luca Cardelli. *Bad engineering properties of object-oriented languages.* (Position paper.) ACM Computing Surveys, 28(4es), Article 150. 1996.

Kriterien:

- Ausführung: wie schnell läuft das Programm?
- Kompilation: wie schnell ist der Compiler?
- Entwicklung im kleinen: wie arbeitet *ein* Programmierer?
- Entwicklung im großen: wie arbeitet ein *Team*?
- Konzepte: wie schwer ist die Sprache zu erlernen?

Aufgaben für OO

- Ausführung: wie schnell läuft das Programm?
  effizienterer Meethoden-Aufruf durch Programm-Analyse
- Kompiliation: wie schnell ist der Compiler?
  separate Kompilation, ohne Zauberei in Interface-Files
- Entwicklung im kleinen: wie arbeitet *ein* Programmierer?
  bessere Typsysteme
- Entwicklung im großen: wie arbeitet ein *Team*?
  Subtyping $<:$ von Subclassing $<\#$ trennen, Typen von Klassen trennen, Polymorphie.
- Konzepte: wie schwer ist die Sprache zu erlernen?
  Ideen orthogonal verbinden