Theoretische Grundlagen der Objektorientierung

Benedikt Meurer

Fachbereich Mathematik
Universität Siegen

Übersicht

Einleitung

Ziel: Untersuchung theoretischer Grundlagen der Objektorientierung

Dazu:

- Entwicklung einer objektorientierten Programmiersprache
- Basierend auf einer (statisch) typsicheren, funktionalen Sprache
- Beweis der Typsicherheit der objektorientierten Programmiersprache

Einleitung

Ziel: Untersuchung theoretischer Grundlagen der Objektorientierung

Dazu:

- Entwicklung einer objektorientierten Programmiersprache
- Basierend auf einer (statisch) typsicheren, funktionalen Sprache
- Beweis der Typsicherheit der objektorientierten Programmiersprache

Vorgehensweise:

- Zunächst einfache funktionale Objekte
- Objektorientierte Kernkonzepte des Typsystems (rekursive Typen, Subtyping)
- Aufbauend darauf schließlich Vererbung (Subclassing)

Funktionale Kernsprache

Kernsprache: ML-Dialekt, ähnlich dem aus „Theorie der Programmierung“ bekannten

Sprachkonzepte:

- Konstanten $\mathit{Const} = \{\mathit{true}, \mathit{false}\} \cup \{+, -, *, <, >, \leq, \geq, =\} \cup \mathbb{Z}$
- Ausdrücke (Programme)

$$
\begin{array}{rcll}
e & ::= & c \in \mathit{Const} \mid x \in \mathit{Var} & \\
 & \mid & e_1\, e_2 & \text{Applikation} \\
 & \mid & \lambda x.e_1 & \text{Funktion} \\
 & \mid & \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 & \text{Lokale Bindung} \\
 & \mid & \textbf{rec}\, x.e_1 & \text{Rekursiver Ausdruck} \\
 & \mid & \textbf{if}\, e_0 \,\textbf{then}\, e_1 \,\textbf{else}\, e_2 & \text{Bedingter Ausdruck}
\end{array}
$$

Übersicht

1. Einleitung
2. Funktionale Objekte
3. Typsicherheit
4. Vererbung
5. Fazit

Funktionale Objekte I

Zunächst: Syntaktische/semantische Erweiterung für (rein funktionale) Objekte (ähnlich `String`-Objekte in Java)

Sprachaspekte

- Klassenlose Objekte
- Datenkapselung (information hiding)
- Objektduplikation (cloning), wie in O'Caml

Abstrakte Syntax:

$$
\begin{aligned}
e &::= \textbf{object}\,(\textit{self}\,)\ r\ \textbf{end} \mid e_1\#m \\
r &::= \textbf{val}\ a = e;\ r_1 \mid \textbf{method}\ m = e;\ r_1 \mid \epsilon
\end{aligned}
$$

Funktionale Objekte II

Beispiel: Zählerobjekt

$$
\begin{array}{l}
\textbf{let}\ \mathit{counter}\ = \\
\quad \textbf{object}\,(\mathit{self}\,) \\
\qquad \textbf{val}\ x = 1; \\
\qquad \textbf{method}\ \mathit{inc} = \{\langle x = x + 1\rangle\}; \\
\qquad \textbf{method}\ \mathit{get} = x; \\
\quad \textbf{end} \\
\textbf{in}\ \mathit{counter}\#\mathit{inc}\#\mathit{inc}\#\mathit{get}
\end{array}
$$

Beispiel: Selbstaufruf

$$(\textbf{object}\,(\mathit{self}\,)\ \textbf{method}\ \mathit{recurse} = \mathit{self}\#\mathit{recurse};\ \ \textbf{end})\#\mathit{recurse}$$

Semantik I

Intuitive Semantik

- In Objekten rechte Seiten von Attributen auswerten

  z.B. **object** (self) **val** $a = 1 + 1$; **val** $b = 2 * 2$; ... **end**
  $\Rightarrow$ **object** (self) **val** $a = 2$; **val** $b = 4$; ... **end**

Semantik I

Intuitive Semantik

- In Objekten rechte Seiten von Attributen auswerten

  z.B. $\textbf{object}\,(\mathit{self})\ \textbf{val}\ a = 1 + 1;\ \textbf{val}\ b = 2 * 2;\ \ldots\ \textbf{end}$
  $\Rightarrow \textbf{object}\,(\mathit{self})\ \textbf{val}\ a = 2;\ \textbf{val}\ b = 4;\ \ldots\ \textbf{end}$

- Methodenaufruf faltet Objekt auf (Objekt für *self* eingesetzt, Duplikationen expandiert)

  z.B. $(\textbf{object}\,(\mathit{self})\ \textbf{method}\ m = \mathit{self}\#n * 4;\ \ldots\ \textbf{end})\#m$
  $\Rightarrow (\textbf{method}\ m = (\textbf{object}\,(\mathit{self})\ \ldots\ \textbf{end})\#n * 4;\ )\#m$

Semantik I

Intuitive Semantik

- In Objekten rechte Seiten von Attributen auswerten

  z.B. $\textbf{object}\,(\mathit{self})\ \textbf{val}\ a = 1+1;\ \textbf{val}\ b = 2*2;\ \ldots\ \textbf{end}$
  $\Rightarrow \textbf{object}\,(\mathit{self})\ \textbf{val}\ a = 2;\ \textbf{val}\ b = 4;\ \ldots\ \textbf{end}$

- Methodenaufruf faltet Objekt auf (Objekt für *self* eingesetzt, Duplikationen expandiert)

  z.B. $(\textbf{object}\,(\mathit{self})\ \textbf{method}\ m = \mathit{self}\#n*4;\ \ldots\ \textbf{end})\#m$
  $\Rightarrow (\textbf{method}\ m = (\textbf{object}\,(\mathit{self})\ \ldots\ \textbf{end})\#n*4;\ )\#m$

- Suche nach letzter Methode mit Namen der Nachricht, dabei Einsetzen der Attributwerte

  z.B. $(\textbf{val}\ a = 42;\ \textbf{method}\ m = a*2;\ \ldots)\#m$
  $\Rightarrow (\textbf{method}\ m = 42*2;\ \ldots)\#m$
  $\Rightarrow 42*2$

Semantik II

Small step Semantik

- Formalisierte Vorgehensweise zur Auswertung
- Programme schrittweise (in „small steps") vereinfachen zu Werten $e \to e_1 \to \ldots \to e_n \in \mathit{Val}$

Small step Berechnung

- terminiert mit Wert („ausgewerteter Ausdruck")
- divergiert (Endlosrekursion)
- bleibt stecken (ungültiger Ausdruck), z.B. $(21 * 2) + \mathit{true}$

Beispiel: Einfache Berechnung

$$\textbf{let}\, f = \lambda x.x * x \,\textbf{in}\, f\,2 \;\to\; (\lambda x.x * x)\,2 \;\to\; 2 * 2 \;\to\; 4$$

Semantik III

Beispiel: small steps für das Senden von Nachrichten

(Send-Eval) $$\frac{e \to e'}{e\#m \to e'\#m}$$

(Send-Unfold) $\textbf{object}\,(\textit{self})\ \omega\ \textbf{end}\#m \to \omega[^{\textbf{object}\,(\textit{self})\ \omega\ \textbf{end}}/_{\textit{self}}]\#m$

(Send-Attr) $(\textbf{val}\ a = v;\ \omega)\#m \to \omega[^v/_a]\#m$

(Send-Skip) $(\textbf{method}\ m' = e;\ \omega)\#m \to \omega\#m$

(Send-Exec) $(\textbf{method}\ m = e;\ \omega)\#m \to e$

Idee:

1. (Send-Eval) bis links ein Objekt, dann (Send-Unfold)
2. (Send-Attr) und (Send-Skip) bis gefunden, dann (Send-Exec)

Übersicht

1. Einleitung
2. Funktionale Objekte
3. Typsicherheit
4. Vererbung
5. Fazit

Typsicherheit

Typsicherheit: *„Well-typed programs don't go wrong!“*

Insbesondere: statische Typsicherheit

- Keine Laufzeittypüberprüfung
- Typüberprüfung ausschließlich auf syntaktischen Informationen (keine exakte Differenzierung $\rightarrow$ Halteproblem)

Typsicherheit

Typsicherheit: „*Well-typed programs don't go wrong!*“

Insbesondere: statische Typsicherheit

- Keine Laufzeittypüberprüfung
- Typüberprüfung ausschließlich auf syntaktischen Informationen (keine exakte Differenzierung → Halteproblem)

Eigenschaft stärker als „Java Typsicherheit“

- Java erlaubt Downcasts
- Java Subtyping-Regeln für Arrays
- ...

→ Java nur „dynamisch typsicher“

Typsystem

Dazu: Typsystem

- explizit (mit Typannotationen, wie Java/C++)
- rekursive Typen
- Subtyping-Polymorphie

Abstrakte Syntax:

$$
\begin{array}{rcll}
\tau & ::= & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} & \text{Primitive Typen} \\
 & \mid & \tau_1 \to \tau_2 & \text{Funktionstypen} \\
 & \mid & \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle & \text{Objekttypen}
\end{array}
$$

Beispiel: $\langle x : \mathbf{int}; y : \mathbf{int} \rangle \to \mathbf{int}$

Rekursive Typen

Beispiel: vereinfachtes Zählerobjekt

$$\textbf{object}\,(\mathit{self} : \tau_z)\ \textbf{val}\ x = 1;\ \textbf{method}\ \mathit{inc} = \{\langle x = x + 1\rangle\};\ \ \textbf{end}$$

Frage: Wie sieht τ_z aus?

Rekursive Typen

Beispiel: vereinfachtes Zählerobjekt

$$\textbf{object}\,(\mathit{self} : \tau_z)\ \textbf{val}\ x = 1;\ \textbf{method}\ \mathit{inc} = \{\langle x = x + 1\rangle\};\ \ \textbf{end}$$

Frage: Wie sieht τ_z aus?

- τ_z muss Objekttyp sein
- τ_z enthält sich selbst als Typ von *inc*
- also zu lösen $\tau_z = \langle \mathit{inc} : \tau_z \rangle$

Rekursive Typen

Beispiel: vereinfachtes Zählerobjekt

$$\textbf{object}\,(\mathit{self} : \tau_z)\ \textbf{val}\ x = 1;\ \textbf{method}\ \mathit{inc} = \{\langle x = x + 1 \rangle\};\ \ \textbf{end}$$

Frage: Wie sieht τ_z aus?

- τ_z muss Objekttyp sein
- τ_z enthält sich selbst als Typ von *inc*
- also zu lösen $\tau_z = \langle \mathit{inc} : \tau_z \rangle$

Dazu:

- rekursive Typen $\quad \tau ::= \mu t.\tau_1 \quad (t \in \mathit{TName})$
- für Zählerobjekt: $\quad \tau_z = \mu z.\langle \mathit{inc} : z \rangle$

Subtyping I

Subtyping: i.d.R. einzige Form von Polymorphie

Beispiel:

$$\textbf{let } f = \lambda o : \langle x : \textbf{int};\ y : \textbf{int}\rangle.\sqrt{o\#x * o\#x + o\#y * o\#y}$$

Allerdings:

- f ohne Subtyping nur auf Objekte mit $\langle x : \textbf{int};\ y : \textbf{int}\rangle$ anwendbar
- aber nicht $\langle x : \textbf{int};\ y : \textbf{int};\ z : \textbf{int}\rangle$
- wider jeglichem Code-Reuse

Offensichtlich: unnötige Einschränkung!

Subtyping II

Beobachtung: Ausdrücke mit „besserem“ Typ problemlos

- Ausdrücke „besseren“ Typs einsetzbar anstelle von Ausdrücken „schlechteren“ Typs (z.B. in Funktion *f*)
- Subtyprelation $\leq$ auf *Type* bestimmt, ob „besser“ ($\rightarrow$ kleiner)
- $\langle\rangle$ ist größter Objekttyp (Informationsgehalt null)

Dazu: Subtyprelation über Regeln definiert, z.B. ohne rek. Typen:

(Sm-Refl) $\quad \beta \leq \beta$ für alle $\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

(Sm-Object)
$$\frac{\tau_i \leq \tau'_j \text{ für alle } i, j \text{ mit } m_i = m'_j}{\langle m_1 : \tau_1; \ldots; m_k : \tau_k\rangle \leq \langle m'_1 : \tau'_1; \ldots; m'_l : \tau'_l\rangle}$$
falls $\{m'_1, \ldots, m'_l\} \subseteq \{m_1, \ldots, m_k\}$

Beweis der Typsicherheit I

Wichtigste Eigenschaft: statische Typsicherheit

Satz (Typesafety): Die Berechnung eines wohlgetypten Ausdrucks bleibt nicht stecken.

Beweis: in zwei Schritten

- Wohlgetyptheit (und Typ) bleibt erhalten bei einem small step ($\rightarrow$ Preservation)
- Wohlgetypter Ausdruck ist entweder fertig ausgewertet oder kann einen small step machen ($\rightarrow$ Progress)

Typesafety dann trivial durch indirekten Beweis

Beweis der Typsicherheit II

Satz (Preservation): Wenn e wohlgetypt mit τ und $e \to e'$, dann ist auch e' wohlgetypt mit τ.

Beweis: Relativ aufwendig, u.a. war zu zeigen:

- Semantische Konzepte (Substitution, Objektauffaltung aka *self*-Substitution, Reiheneinsetzung, etc.) sind typerhaltend
- Ausdrücke haben je nach Form eingeschränkten Bereich möglicher Typen

Dann Preservation induktiv leicht zu beweisen

Dabei: etliche Roundtrips für Sprachdefinitionen, um Preservation beweisen zu können

Beweis der Typsicherheit III

Satz (Progress): Wenn e wohlgetypt, dann entweder $e \to e'$ oder e fertig.

Beweis: Verhältnismäßig einfache Induktion, mit Prämissen der Preservation.

Beweis der Typsicherheit III

Satz (Progress): Wenn e wohlgetypt, dann entweder $e \rightarrow e'$ oder e fertig.

Beweis: Verhältnismäßig einfache Induktion, mit Prämissen der Preservation.

Insgesamt: Typsicherheit für small step Semantik

- eingeschränkt übertragbar auf andere Semantiken (Äquivalenzbeweis)
- big step Semantik interessant für Implementierung
- für OO Sprachen bisher nicht bekannt (O'Caml nur unvollständig formalisiert)

Übersicht

Vererbung

Vererbung: Sprache um Klassen erweitert

Interessant dabei:

- Kein information hiding bei Klassen ($\rightarrow$ Klassentypen)
- Vererbung bedingt eine Form der Polymorphie für *self*-Typ (hier: Subtyping-Polymorphie wie Java/C++)
- O'Caml benutzt ML-Polymorphie (echt mächtiger als Subtyping-Polymorphie)
- Kein Subtyping auf Klassentypen

Typsicherheit: Wie zuvor, allerdings Preservation aufwändiger durch Vererbung und *self*-Subtyping

Übersicht

Fazit:

- OO Konzepte möglich in statisch typsicherer Sprache
- Ergebnisse (eingeschränkt) anwendbar auf praktische Sprachen ($\rightarrow$ Formalisierung notwendig)
- Theorie mit Typinferenz/ML-Polymorphie und/oder imperativen Konzepten ausstehend
- Forschung an Typsystemen notwendig (Optimum leider unentscheidbar)