Concurrent Object-Oriented Programming

Gul Agha, Ian Mason, Scott Smith, Carolyn Talcott
Vortrag: Marcel Kyas

4. Februar 1999

Literatur

[Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[AMST92] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In *Lecture Notes in Computer Science*, pages 565–579. Springer Verlag, 1992.

[AMST93] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal on Functional Programming*, January 1993.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society student texts*. Cambridge University Press, 1986.

Motivation

1. Verstärkte Interaktion von Benutzer-Prozessen (z.B. X-Windows, Qt)
2. Netzwerke von Workstations als kostengünstige Parallelrechner (z.B. distributed.net, PVM, Linux Beowulf)
3. Mehrprozessorrechner sind kostengünstig geworden.

$\rightarrow$ parallele Berechnungen in *distributed systems* [Agh90].

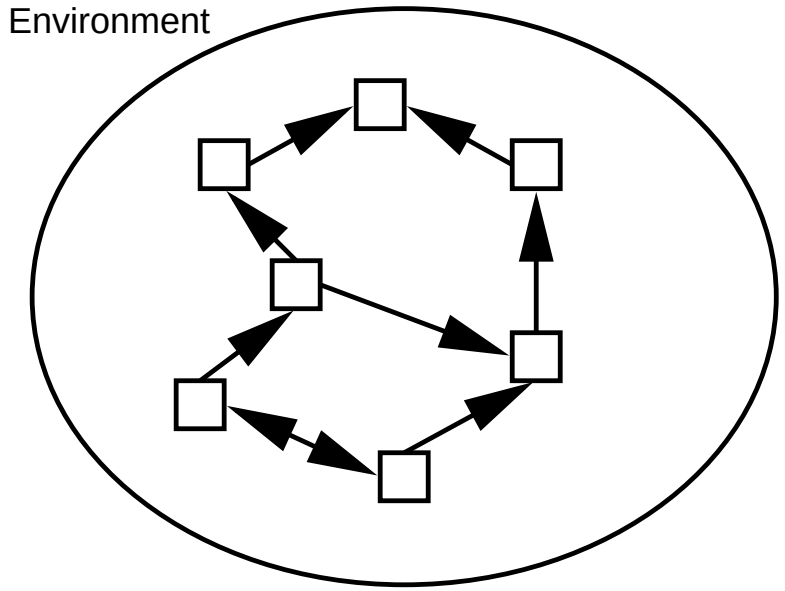

Abbildung 1: Geschlossenes verteiltes System

Offene verteilte Systeme

Das System nimmt Eingaben aus seiner Umgebung an.

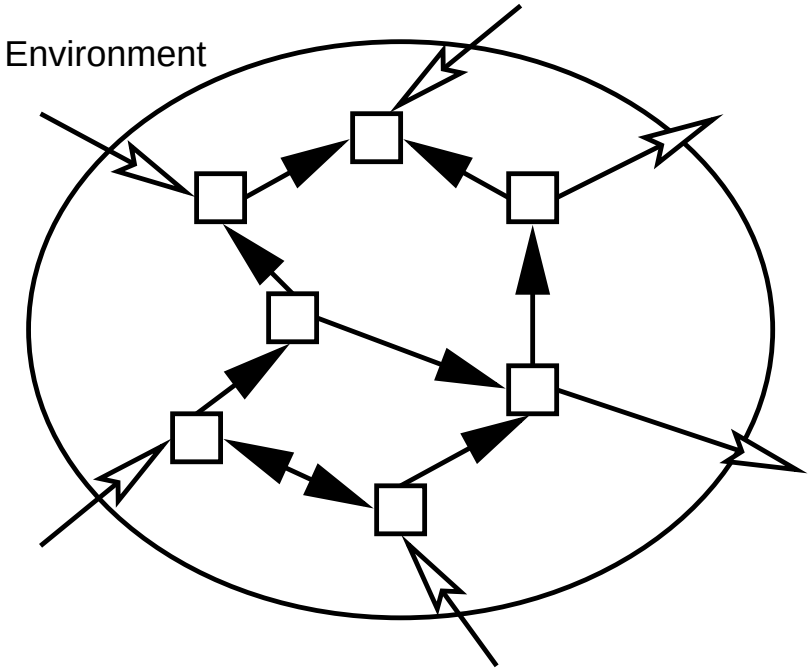

Abbildung 2: Offenes verteiltes System

Definition 1. *Objekte* kapseln Daten und Operationen in eine einzelne berechnende Einheit.

Hier Identifikation von Prozessen mit Objekten.

Definition 2. *Aktoren* sind selbst erhaltende, interaktive, unabhängige Komponenten eines berechnenden Systems, die über asynchrones *message passing* kommunizieren. Sie können dynamisch erschaffen werden und ihre Topologie kann sich ebenso verändern.

Exkurs 1: λ-Kalkül

Definition der Semantik im λ-Kalkül.

Definition 3 (λ-Terme). Sei $\mathbf{V}$ eine endliche Menge von Variablen, $\mathbf{C}$ eine endliche Menge von Konstanten.

1. Alle Variablen und Konstanten sind λ-Terme (Atome).
2. Sind M und N λ-Terme, dann ist (MN) ein λ-Term (Applikation).
3. Ist M ein λ-Term und x eine Variable, dann ist $(\lambda x.M)$ ein λ-Term.

Bemerkung 1. Lisp und Scheme sind Implementationen eines erweiterten λ-Kalküls.

Definition 4. $[y/x]$ bezeichne die *gebundene Umbenennung* von x in y. $\mathrm{FV}(M)$ sei die Menge aller freien Variablen des Terms M.

Definition 5.

$$(\lambda x.M) \rightarrow_\alpha (\lambda y.[y/x]M)$$

$$(\lambda x.M)N \rightarrow_\beta [N/x]M$$

$$(\lambda x.Mx) \rightarrow_\eta M, \text{ falls } x \notin \mathrm{FV}(M)$$

$\rightarrow_{\beta\eta}$ bezeichne eine beliebige der obigen Transformationen. $\rightarrow_i^*$ mit $i \in \{\alpha, \beta, \eta, \beta\eta\}$ ist die transitive Hülle auf $\rightarrow_i$.

λ-Kalkül (Fortsetzung)

Satz 1 (Church-Rosser). *Es gilt:*

$$\forall P.\forall M.\forall N.P \to^*_{\beta\eta} M \wedge P \to^*_{\beta\eta} N \Longrightarrow \exists T.M \to^*_{\beta\eta} T \wedge N \to^*_{\beta\eta} T$$

Beweis. Siehe [HS86, S. 313–322]. □

Definition 6 (Y-Kombinator, Alan M. Turing).

$$Z := (\lambda z.(\lambda x.x(zzx)))$$
$$Y := ZZ$$

Satz 2 (Fixpunkt-Satz). *Es gilt:* $Yx \to^*_\beta x(Yx)$

Beweis.

$$\begin{array}{rl} & (\lambda z.(\lambda x.x(zzx)))Zx \\ \to_\beta & ([Z/z](\lambda x.x(zzx)))x \\ \equiv & (\lambda x.x(ZZx))x \\ \to_\beta & x(ZZx) \\ \equiv & x(Yx) \end{array}$$

□

Primitive Aktor-Handlungen

Sei b ein λ-Term, genannt *Verhalten*.

`send`(a, v) sendet eine Botschaft v an einen Aktoren a.

`newadr`$()$ erschafft einen neuen Aktoren ohne Verhalten.

`initb`(a, b) initialisiert ein neu erschaffenen Aktoren a mit dem Verhalten b.

`become`(b) ändert das Verhalten eines Aktoren in b und erschafft einen neuen anonymen Aktoren, der die restlichen Berechnungen ausführt.

Der neue Aktor kann Nachrichten versenden oder neue Aktoren erschaffen, aber nie Nachrichten empfangen, da seine Adresse niemals ermittelt werden kann.

- Zustand eines Aktoren läßt sich aus seinem initialem Zustand und der Geschichte seiner Kommunikation ermitteln.
- Versenden nur von *elementaren* Werten, nicht von λ-Termen.
- Durch ändern des Verhaltens: Serialisations-Mechanismus, der einen trivialen *pipelining*-Mechanismus unterstützt.

Beispiel: Mergesort

```
(define mergesort
                    (λ[list len]
                    (if (= len 1)
                    list
                    (merge(mergesort(first − half [list len]))
                          (mergesort(second − half [list len]))))))
```

Abbildung 3: Code eines Mergesort

Funktionsaufrufe werden nebenläufig ausgeführt. Erzeugung von Aktoren geschieht hier implizit.

Join Continuations: Nebenläufig ausführbare Programmteile werden so ausgeführt. Erzeugter Aktor wird Join Continuation genannt.

Dies ist ein Beispiel für *divide-and-conquer concurrency*. Andere Formen: *pipeline concurrency* (z.B. Sieb des Eratosthenes), *cooperative problem solving concurrency* (z.B. Simulationen).

Satz 3. *Mergesort hat eine Laufzeit in* $O(n \log n)$.

Beweisführungen über Aktoren

Ausführliche Darstellung in [AMST93]!

Definition 7 (Konfiguration). Sei α eine Aktor-Abbildung, μ eine Multi-Menge von Nachrichten, ρ eine Menge von Empfängern und χ eine Menge von externen Aktoren. Eine Konfiguration wird notiert als:

$$\langle \alpha | \mu \rangle_{\chi}^{\rho}$$

Wir fordern weiterhin:

1. $\rho \subseteq \mathrm{Dom}\alpha$ und $\mathrm{Dom}\alpha \cap \chi = \emptyset$.
2. $a \in \mathrm{Dom}\alpha \Longrightarrow \mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, und $\alpha(a) = (?_{a'}) \Longrightarrow a' \in \mathrm{Dom}\alpha$.
3. $< a \Leftarrow v > \in \mu \Longrightarrow \mathrm{FV}(v) \cup \{a\} \subseteq A \cup \chi$.

Definition 8 (Aktor-Zustände). Sei a ein Aktor.

$(?'_a)_a$ bezeichne einen von a' erzeugten uninitialisierten Aktoren.

$(b)_a$ bezeichne einen Aktoren, der eine Nachricht im Verhalten b zu empfangen.

$[e]_a$ bezeichne einen Aktoren, der den Ausdruck e auswertet.

Definition 9. Eine Nachricht v an den Aktoren a wird als $< a \Leftarrow v >$ notiert. v ist ein *elementarer Wert* und kein λ-Term!

Semantik

Definition 10 (Semantik). $\rightarrow$ bezeichne einen einfachen Schritt in einer Aktor-Konfiguration.

$$e \rightarrow_{\beta\eta} e' \Rightarrow \langle \alpha, [e]_a | \mu \rangle_\chi^\rho \rightarrow \langle \alpha, [e']_a | \mu \rangle_\chi^\rho \quad (1)$$

$$\langle \alpha, [\texttt{newadr}()]_a | \mu \rangle_\chi^\rho \rightarrow \langle \alpha, [a']_a, (?_a)_{a'} | \mu \rangle_\chi^\rho \quad (2)$$

$$\langle \alpha, [\texttt{initb}(a', b)]_a, (?_a)_{a'} | \mu \rangle_\chi^\rho \rightarrow \langle \alpha, [\texttt{nil}]_a, (b)_{a'} | \mu \rangle_\chi^\rho \quad (3)$$

$$\langle \alpha, [\texttt{become}(b)]_a | \mu \rangle_\chi^\rho \rightarrow \langle \alpha, (b)_a, [\texttt{nil}]_\_ | \mu \rangle_\chi^\rho \quad (4)$$

$$\langle \alpha, [\texttt{send}(a', v)]_a | \mu \rangle_\chi^\rho \rightarrow \langle \alpha, [\texttt{nil}]_a | \mu, < a' \Leftarrow v > \rangle_\chi^\rho \quad (5)$$

$$\langle \alpha | < a \Leftarrow v >, \mu \rangle_\chi^\rho \rightarrow \langle \alpha, [b(v)]_a | \mu \rangle_\chi^\rho \quad (6)$$

$$\langle \alpha | < a \Leftarrow v >, \mu \rangle_\chi^\rho \rightarrow \langle \alpha | \mu \rangle_\chi^{\rho \cup \{\mathrm{FV}(v) \cap \mathrm{Dom}\alpha\}} \quad (7)$$

$$\langle \alpha | \mu \rangle_\chi^\rho \rightarrow \langle \alpha | < a \Leftarrow v >, \mu \rangle_{\chi \cup \{\mathrm{FV}(v) \cap \mathrm{Dom}\alpha\}}^\rho \quad (8)$$

In (3) ist $a' \notin \mathrm{Dom}\alpha$, also eine neue Variable. In (7) gilt weiter $a \in \chi$. In (8) gilt $a \in \rho$ und $\mathrm{FV}(v) \cap \mathrm{Dom}\alpha \subseteq \rho$.

$\rightarrow^*$ ist die transitive, reflexive Hülle von $\rightarrow$.

Damit: Transitionssystem auf Aktor-Konfigurationen.

In [AMST93] benutzte Technik: *operational bisimulation*.

Berechnungen

Definition 11. Ist κ eine Konfiguration, dann ist der *Berechnungsbaum* $\mathcal{T}(\kappa)$ die Menge aller Folgen von Transitionen mit $\kappa_i \rightarrow \kappa_{i+1}$ für $i \in \mathbb{N}$ und $\kappa = \kappa_0$.

Definition 12 (Komponierbar). Seien $\kappa_i = \langle \alpha_i | \mu_i \rangle_{\chi_i}^{\rho_i}$ mit $i \in \{1, 2\}$ zwei Aktor-Konfiguration. Wir nennen sie *komponierbar*, wenn $\mathrm{Dom}\alpha_1 \cap \mathrm{Dom}\alpha_2 = \emptyset$, $\chi_1 \cap \mathrm{Dom}\alpha_2 \subseteq \rho_2$ und $\chi_2 \cap \mathrm{Dom}\alpha_1 \subseteq \rho_1$ gilt.

Definition 13 (Komposition, Dekomposition). Die *Komposition* $\kappa_1 \| \kappa_2$ zweier Konfigurationen $\kappa_i = \langle \alpha_i | \mu_i \rangle_{\chi_i}^{\rho_i}$, $i \in \{1, 2\}$, ist:

$$\kappa_1 \cup \kappa_2 = \langle \alpha_1 \cup \alpha_2 | \mu_1 \cup \mu_2 \rangle_{(\chi_1 \cup \chi_2) \setminus (\rho_1 \cup \rho_2)}^{(\rho_1 \cup \rho_2)},$$

(κ_1, κ_2) ist eine *Dekomposition* von κ, falls κ_1 und κ_2 komponierbar sind und $\kappa = \kappa_1 \| \kappa_2$ gilt.

Lemma 1. *Seien* $\kappa_i = \langle \alpha_i | \mu_i \rangle_{\chi_i}^{\rho_i}$, $i \in \{1, 2, 3\}$, *paarweise komponierbar und* $\kappa_0 = \langle \emptyset | \emptyset \rangle_{\emptyset}^{\emptyset}$. *Dann gilt:*

$$\kappa_1 \| \kappa_2 = \kappa_2 \| \kappa_1$$

$$\kappa_0 \| \kappa_1 = \kappa_1$$

$$(\kappa_1 \| \kappa_2) \| \kappa_3 = \kappa_1 \| (\kappa_2 \| \kappa_3)$$

Berechnungen, Kompositionen

Satz 4 (Komposition offener Konfigurationen). *Es gibt eine binäre Operation $\mathcal{M}$ auf Berechnungsbäumen, so daß, wenn $\kappa_i, i \in \{1, 2\}$ komponierbar sind, gilt:*

$$\mathcal{T}(\kappa_1 \cup \kappa_2) = \mathcal{M}(\mathcal{T}(\kappa_1), \mathcal{T}(\kappa_2))$$

Beweis. Einfache Übung. □

Abschließend:

- Unerwähnt:
  - Äquivalenz von Konfigurationen.
  - Beweistheorie.
  - Gesetze der elementaren Aktor-Operationen.
  - Vererbung und Modularität.
- Hauptideen:
  - Kapselung von Operationen und Zuständen.
  - Abstraktion der Kommunikations- und Synchronisations-Mechanismen.
  - Funktionaler Ansatz.