Monaden

Frank-André Rieß

5. Juni 2007

Motivation

Wofür benötigt man Monaden?

Motivation

Wofür benötigt man Monaden?

- Imperative Programmiersprachen:

- Funktionale Programmiersprachen:

Motivation

Wofür benötigt man Monaden?

- Imperative Programmiersprachen:
  - Zugriff auf Variablen/Referenzen – implizite Zustandsveränderung möglich

- Funktionale Programmiersprachen:

Motivation

Wofür benötigt man Monaden?

- Imperative Programmiersprachen:
  - Zugriff auf Variablen/Referenzen – implizite Zustandsveränderung möglich
  - Exceptions etc. – Beeinflussung des Kontrollflusses zur Fehlerbehandlung
- Funktionale Programmiersprachen:

Motivation

Wofür benötigt man Monaden?

- Imperative Programmiersprachen:
  - Zugriff auf Variablen/Referenzen – implizite Zustandsveränderung möglich
  - Exceptions etc. – Beeinflussung des Kontrollflusses zur Fehlerbehandlung
- Funktionale Programmiersprachen:
  - nur Zugriff auf Argumente einer Funktion – Zustandsveränderungen müssen durch explizite Änderung eines „Zustandsobjektes“ simuliert werden

Motivation

Wofür benötigt man Monaden?

- Imperative Programmiersprachen:
  - Zugriff auf Variablen/Referenzen – implizite Zustandsveränderung möglich
  - Exceptions etc. – Beeinflussung des Kontrollflusses zur Fehlerbehandlung
- Funktionale Programmiersprachen:
  - nur Zugriff auf Argumente einer Funktion – Zustandsveränderungen müssen durch explizite Änderung eines „Zustandsobjektes“ simuliert werden
  - Fehlerbehandlung nur explizit über (z.B. zusammengesetzten) Rückgabewert oder Komplettabbruch

Motivation

Wünschenswert: Möglichkeit, das Zustandsobjekt (etc.) „unsichtbar“ zu machen..

Motivation

Wünschenswert: Möglichkeit, das Zustandsobjekt (etc.) „unsichtbar“ zu machen..

- ..durch Einbau unreiner Züge in die Programmiersprache

Motivation

Wünschenswert: Möglichkeit, das Zustandsobjekt (etc.) „unsichtbar“ zu machen..

- ..durch Einbau unreiner Züge in die Programmiersprache – dadurch aber Verlust der referenziellen Transparenz

Motivation

Wünschenswert: Möglichkeit, das Zustandsobjekt (etc.) „unsichtbar“ zu machen..

- ..durch Einbau unreiner Züge in die Programmiersprache – dadurch aber Verlust der referenziellen Transparenz (der Wert eines Ausdrucks ist nicht mehr unabhängig vom Zeitpunkt seiner Auswertung)

Motivation

Wünschenswert: Möglichkeit, das Zustandsobjekt (etc.) „unsichtbar“ zu machen..

- ..durch Einbau unreiner Züge in die Programmiersprache – dadurch aber Verlust der referenziellen Transparenz (der Wert eines Ausdrucks ist nicht mehr unabhängig vom Zeitpunkt seiner Auswertung)
- ..durch geeignete Abstraktion, die den Zustand weitestgehend verbirgt

Zustand in funktionaler Programmierung

Funktion ohne Zustand:

$$x \longrightarrow \boxed{f} \longrightarrow y$$

$$f : \alpha \rightarrow \beta$$

Zustand in funktionaler Programmierung

Funktion ohne Zustand:

$$x \longrightarrow \boxed{f} \longrightarrow y$$

$$f : \alpha \rightarrow \beta$$

Funktion mit explizitem Zustand:

$$\begin{pmatrix} x \\ z_{\text{alt}} \end{pmatrix} \longrightarrow \boxed{f_{\text{Zustand}}} \longrightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix}$$

$$f_{\text{Zustand}} : (\alpha, s) \rightarrow (\beta, s)$$

Zustand in funktionaler Programmierung

Funktion mit explizitem Zustand:

$$\begin{pmatrix} x \\ z_{\text{alt}} \end{pmatrix} \longrightarrow \boxed{f_{\text{Zustand}}} \longrightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix}$$

$$f_{\text{Zustand}} : (\alpha, s) \longrightarrow (\beta, s)$$

Zustand in funktionaler Programmierung

Funktion mit explizitem Zustand:

$$\begin{pmatrix} x \\ z_{\text{alt}} \end{pmatrix} \longrightarrow \boxed{f_{\text{Zustand}}} \longrightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix}$$

$$f_{\text{Zustand}} : (\alpha, s) \longrightarrow (\beta, s)$$

Currying..

Zustand in funktionaler Programmierung

Funktion mit explizitem Zustand:

$$\begin{pmatrix} x \\ z_{\text{alt}} \end{pmatrix} \longrightarrow \boxed{f_{\text{Zustand}}} \longrightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix}$$

$$f_{\text{Zustand}} : (\alpha, s) \longrightarrow (\beta, s)$$

Currying..

$$x \longrightarrow \boxed{f'_{\text{Zustand}}} \longrightarrow \left(z_{\text{alt}} \rightarrow \boxed{f'_{\text{Zustand}} x} \rightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix} \right)$$

$$f'_{\text{Zustand}} : \alpha \longrightarrow s \longrightarrow (\beta, s)$$

Zustand in funktionaler Programmierung

$$x \longrightarrow \boxed{f'_{\text{Zustand}}} \longrightarrow \left(z_{\text{alt}} \rightarrow \boxed{f'_{\text{Zustand}} x} \rightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix} \right)$$

$$f'_{\text{Zustand}} : \alpha \longrightarrow s \longrightarrow (\beta, s)$$

Zustand in funktionaler Programmierung

$$x \longrightarrow \boxed{f'_{\text{Zustand}}} \longrightarrow \left(z_{\text{alt}} \rightarrow \boxed{f'_{\text{Zustand}}\, x} \rightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix} \right)$$

$$f'_{\text{Zustand}} : \alpha \longrightarrow s \longrightarrow (\beta, s)$$

Zustand in funktionaler Programmierung

$$x \longrightarrow \boxed{f'_{\text{Zustand}}} \longrightarrow \left(z_{\text{alt}} \rightarrow \boxed{f'_{\text{Zustand}}\, x} \rightarrow \begin{pmatrix} y \\ z_{\text{neu}} \end{pmatrix} \right)$$

$$f'_{\text{Zustand}} : \alpha \longrightarrow s \longrightarrow (\beta, s)$$

Zustand in funktionaler Programmierung

Sei $\tau_\beta = s \longrightarrow (\beta, s)$ der Typ aller zustandsabhängigen Berechnungen, die ein Ergebnis von Typ β liefern.

Zustand in funktionaler Programmierung

Sei $\tau_\beta = s \longrightarrow (\beta, s)$ der Typ aller zustandsabhängigen Berechnungen, die ein Ergebnis von Typ β liefern.

Man kann sich nun „Getter“ und „Setter“ definieren:

Zustand in funktionaler Programmierung

Sei $\tau_\beta = s \to (\beta, s)$ der Typ aller zustandsabhängigen Berechnungen, die ein Ergebnis von Typ β liefern.

Man kann sich nun „Getter“ und „Setter“ definieren:

$$\text{get} : s \to (s, s) = \tau_s$$

Zustand in funktionaler Programmierung

Sei $\tau_\beta = s \to (\beta, s)$ der Typ aller zustandsabhängigen Berechnungen, die ein Ergebnis von Typ β liefern.

Man kann sich nun „Getter“ und „Setter“ definieren:

$$\text{get} : s \to (s, s) = \tau_s$$
$$\text{get} = \lambda z \to (z, z)$$

Zustand in funktionaler Programmierung

Sei $\tau_\beta = s \to (\beta, s)$ der Typ aller zustandsabhängigen Berechnungen, die ein Ergebnis von Typ β liefern.

Man kann sich nun „Getter“ und „Setter“ definieren:

$$\text{get} : s \to (s, s) = \tau_s$$
$$\text{get} = \lambda z \to (z, z)$$

$$\text{set} : s \to s \to ((), s) = s \to \tau_{()}$$

Zustand in funktionaler Programmierung

Sei $\tau_\beta = s \to (\beta, s)$ der Typ aller zustandsabhängigen Berechnungen, die ein Ergebnis von Typ β liefern.

Man kann sich nun „Getter“ und „Setter“ definieren:

$$\text{get} : s \to (s, s) = \tau_s$$
$$\text{get} = \lambda z \to (z, z)$$

$$\text{set} : s \to s \to ((), s) = s \to \tau_()$$
$$\text{set}\, z = \lambda \_ \to ((), z)$$

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

return : $\alpha \to \tau_\alpha$

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

$$\text{return} : \alpha \to \tau_\alpha$$

$$\text{return}\, x = \lambda\, z \to (x, z)$$

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

$$\text{return} : \alpha \to \tau_\alpha$$

$$\text{return}\, x = \lambda\, z \to (x, z)$$

Zwei Zustandsberechnungen hintereinander ausführen:

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

$$\text{return} : \alpha \to \tau_\alpha$$

$$\text{return}\, x = \lambda\, z \to (x, z)$$

Zwei Zustandsberechnungen hintereinander ausführen:

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

$$\text{return} : \alpha \to \tau_\alpha$$
$$\text{return}\, x = \lambda z \to (x, z)$$

Zwei Zustandsberechnungen hintereinander ausführen:

$$\text{bind} : \tau_\alpha \to (\alpha \to \tau_b eta) \to \tau_b eta$$

Zustand in funktionaler Programmierung

$$\tau_\beta = s \to (\beta, s)$$

Werte in eine Zustandsberechnung hineinbringen:

$$\text{return} : \alpha \to \tau_\alpha$$
$$\text{return}\, x = \lambda\, z \to (x, z)$$

Zwei Zustandsberechnungen hintereinander ausführen:

$$\text{bind} : \tau_\alpha \to (\alpha \to \tau_b eta) \to \tau_b eta$$
$$f\ \text{bind}\ g = \lambda\, z \to \text{let}\ (x, z') = f\ z\ \text{in}\ g\ x\ z'$$