

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 10.11.09:
Monadentransformer

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Heute gibt's:

- ▶ Eine Monade ist **gut**, mehrere Monaden sind **besser**
- ▶ Kombination von Monaden
- ▶ **Monadentransformer**

Fahrplan

- ▶ Teil I: Monaden und fortgeschrittene Typen
 - ▶ Einführung, Wiederholung
 - ▶ Zustände, Zustandsübergänge und IO
 - ▶ Reader/Writer, Nichtdeterminismus, Ausnahmen
 - ▶ Monadentransformer
- ▶ Teil II: Fortgeschrittene Datenstrukturen
- ▶ Teil III: Nebenläufigkeit
- ▶ Teil IV: The Future of Programming

Kombination von Monaden: Das Problem

- ▶ Gegeben zwei **Monaden**:

```
class Monad m1 where ...  
  
class Monad m2 where ...
```

- ▶ Es gelten weder

```
instance Monad (m1 (m2 a))  
instance Monad (m2 (m1 a))
```

- ▶ Problem: **Monadengesetze** gelten nicht.

Monadentransformer

- ▶ Monadentransformer:

- ▶ Erweiterbare Monade

- ▶ Monade mit Loch

```
newtype M a = ...
```

```
instance Monad M where
```

```
  f >>= g = ...
```

```
  return x = ...
```

⇒

```
newtype M m a = ...
```

```
instance Monad m ⇒ instance Monad M m
```

```
  f >>= g = ...
```

```
  return x = ...
```

Monadentransformer: Beispiele

1. Beispiel: Zustandsmonadentransformer

```
type StateT s m a = s → m (s, a)
```

- ▶ Zustandsbasierte Berechnungen in einer anderen Monade m
- ▶ `StateT s Identity` ist Zustandsmonade

2. Beispiel: Fehlermonadentransformer

```
type ErrorT e m a = m (Either e a)
```

- ▶ Fehlerbehaftete Berechnungen in einer anderen Monade m
- ▶ `ErrorT e Identity` ist Fehlermonade

Reihenfolge beachten!

- ▶ Kombination von State und Error
- ▶ Erst **Zustand**, dann **Fehler**:

```
type ErrorState s a =  
    ErrorT (StateT s Id) = Either String (s → (s , a))
```

- ▶ Berechnung $m :: \text{ErrorState } s \ a$: Fehler oder zustandsbehaftet
 - ▶ E.g. Fehler in Haskell
 - ▶ Erst **Fehler**, dann **Zustand**:
- ```
type StateError s a = s → (s , Either String a)
```
- ▶ Berechnung  $m :: \text{StateError } s \ a$ : Immer zustandsbehaftet, Resultat Fehler oder normal
  - ▶ E.g. Fehler in imperativen Sprachen

# Standardtransformer

| Standard-Monade | Transformer | Standard-Typ             | Transformierter Typ          |
|-----------------|-------------|--------------------------|------------------------------|
| Error           | ErrorT      | <code>Either e a</code>  | <code>m (Either e a)</code>  |
| State           | StateT      | <code>s → (a,s)</code>   | <code>s → m (a,s)</code>     |
| Reader          | ReaderT     | <code>r → a</code>       | <code>r → m a</code>         |
| Writer          | WriterT     | <code>(a,w)</code>       | <code>m (a,w)</code>         |
| Cont            | ContT       | <code>(a → r) → r</code> | <code>(a → m r) → m r</code> |

Quelle: <http://www.haskell.org/all`about`monads/>

# Fallbeispiel: ein Modularer Interpreter

- ▶ Ziel: Interpreter für eine einfache imperative Sprache
- ▶ Modularer Aufbau:
  1. Nur Zustand
  2. Ausgabe
  3. Eingabe
  4. Fehler und Fehlerbehandlung

# Zusammenfassung

- ▶ **Warum** Monaden kombinieren?
  - ▶ Typ definiert **Effekt**
  - ▶ E.g. `WriterT` — nur Logging, kein Zustand
  - ▶ Pragmatisch: die `IO`-Monade (imperativ)
- ▶ **Vorteile** Monadentransformer:
  - ▶ Erlauben modulare Kombination von Monaden
  - ▶ Standard-Bücherei (`Monad Template Library`) bietet Standard-Monaden als praktischen Bausatz
- ▶ **Nachteile** Monadentransformer:
  - ▶ Für neue Transformer **Kombination** mit **allen** anderen zu bedenken!
  - ▶ Nicht **kompositional**
- ▶ Nächste Woche: ?