

# *Monaden*

---

Proseminar  
Perlen der Informatik (WS1999)  
bei Prof. Dr. T. Nipkow

Vortrag von:  
**Julian Mehnle**

# *Reine Sprachen*

---

Reine Sprachen (z.B. Haskell, Miranda), zeichnen sich aus durch:

- rein expliziten Datenfluss
- leichte Beweisbarkeit von Algorithmen und Programmen
- Lazy Evaluation

Unreine Sprachen (z.B. Scheme, Standard ML) bieten dafür aber oft zusätzliche Features:

- Exceptions
- Zuweisungen / implizite Zustandsänderungen
- Möglichkeit einer kompakteren Programmierung
- höhere Effizienz

# Monaden

---

- In den 60ern in der Kategorientheorie erfunden
- in den 70ern erstmals auf funktionale Programmierung angewendet
- Philip Wadler (bei Bell Labs (Lucent Technologies), damals University of Glasgow) trug wesentlich zur Entwicklung und Beschreibung von Monaden bei, ebenfalls zur Entwicklung von Haskell und der Integration von Monaden in letzteres.
- ermöglichen die Umsetzung vieler "unreiner" Konzepte in reine, funktionale Sprachen, ohne die Vorteile der letzteren zunichte zu machen: z.B. Exceptions, Verwaltung globaler Zustände, oder Ein-/Ausgabe. Ausserdem können Monaden zur Strukturierung von funktionalen Programmen dienen.

# Ein einfacher Auswerter

---

Terme der Bauart (*Div* (*Div* (*Con* 1972) (*Con* 2)) (*Con* 23))  
lassen sich so darstellen:

**data** *Term* = *Con Int* | *Div Term Term*

Der passende Auswerter in Haskell:

```
eval           :: Term  
eval (Con a)  = a  
eval (Div t u) = eval t ÷ u
```

# Der Auswerter mit Exceptions

---

```
data M a      = Raise Exception | Return a
type Exception = String

eval          :: Term → M Int
eval (Con a) = Return a
eval (Dio t u) = case eval t of
    Raise e → Raise e
    Return a → case eval u of
        Raise e → Raise e
        Return b → if b == 0
            then Raise "divide by zero"
            else Return (a ÷ b)
```

# Der Auswerter mit globalem Zustand

---

**type**  $M\ a$       =  $State \rightarrow (a, State)$   
**type**  $State$     =  $Int$

$eval$             ::  $Term \rightarrow M\ Int$   
 $eval\ (Con\ a)\ x$     =  $(a, x)$   
 $eval\ (Div\ t\ u)\ x$     = **let**  $(a, y) = eval\ t\ x$  **in**  
                          **let**  $(b, z) = eval\ u\ y$  **in**  
                           $(a \div b, z + 1)$

# Definition einer Monade

---

Statt einer Funktion vom Typ  $a \rightarrow b$  wird eine vom Typ  $a \rightarrow M a$  verwendet, wobei  $M$  eine Nebenwirkung der Funktion darstellt (z.B. Fehlerbehandlung oder erhöhen eines globalen Zählers).

Eine **Monade** ist ein Tripel  $(M, \text{unit}, \star)$  aus dem in obiger Funktion verwendeten  $M$  und folgenden zwei sog. Monadenoperationen:

$$\begin{aligned} \text{unit} &:: a \rightarrow M a \\ (\star) &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \end{aligned}$$



# Der monadische Auswerter

---

**type**  $M\ a$        $=\ a$

*unit*       $::\ a \rightarrow M\ a$

*unit*  $a$        $=\ a$

( $\star$ )       $::\ M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$

$a\ \star\ k$        $=\ k\ a$

*eval*       $::\ Term \rightarrow M\ Int$

*eval* (*Con*  $a$ )       $=\ unit\ a$

*eval* (*Div*  $t\ u$ )       $=\ eval\ t\ \star\ \lambda a. eval\ u\ \star\ \lambda b. unit\ (a\ \div\ b)$

Einsetzen und Vereinfachen liefert den einfachen Auswerter.

# Monadischer Auswerter mit Exceptions (Monadendefinition)

---

**data**  $M\ a$  = *Raise Exception* | *Return a*

**type** *Exception* = *String*

*unit* ::  $a \rightarrow M\ a$

*unit a* = *Return a*

( $\star$ ) ::  $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$

**$m \star k$**

*Raise e*  $\rightarrow$  *Raise e*

*Return a*  $\rightarrow$  *k a*

*raise* :: *Exception*  $\rightarrow$   $M\ a$

*raise e* = *Raise e*



# Monadischer Auswarter mit globalem Zustand

## (Monadendefinition)

---

```
type  $M a$  =  $State \rightarrow (a, State)$ 
type  $State$  =  $Int$ 

unit
unit  $a$  ::  $a \rightarrow M a$ 
      =  $\lambda x.(a, x)$ 

( $\star$ )
 $m \star k$  ::  $M a \rightarrow (a \rightarrow M b) \rightarrow M b$ 
      =  $\lambda x. \mathbf{let} (a, y) = m x \mathbf{ in}$ 
         $\mathbf{let} (b, z) = k a y \mathbf{ in}$ 
           $(b, z)$ 

tick
tick ::  $M ()$ 
      =  $\lambda x. ((), x + 1)$ 
```

# Monadischer Auswerter mit globalem Zustand (Auswerter)

---

$eval$   $:: Term \rightarrow M Int$   
 $eval (Con a)$   $= unit a$   
 $eval (Div t u)$   $= eval t \star \lambda a. eval u \star \lambda b. unit (a \div b)$

wird zu:

$eval$   $:: Term \rightarrow M Int$   
 $eval (Con a)$   $= unit a$   
 $eval (Div t u)$   $= eval t \star \lambda a. eval u \star \lambda b. tick \star \lambda(). unit (a \div b)$

# Monadengesetze

---

— Links-Einheit:

$$\text{unit } a \star \lambda b.n = n[a/b]$$

— Rechts-Einheit:

$$m \star \lambda a. \text{unit } a = m$$

— Assoziativität:

$$m \star (\lambda a.n \star \lambda b.o) = (m \star \lambda a.n) \star \lambda b.o$$



# List Comprehensions

---

$$\begin{aligned} [ \text{sqr } a \mid a \leftarrow [1, 2, 3] ] &= [1, 4, 9] \\ [ (a, b) \mid a \leftarrow [1, 2], b \leftarrow \text{"list"} ] &= [(1, 'l'), (1, 'i'), (1, 's'), (1, 't'), \\ &\quad (2, 'l'), (2, 'i'), (2, 's'), (2, 't')] \end{aligned}$$

Dies lässt sich wie folgt in Monadenoperationen übersetzen:

$$\begin{aligned} [ t \mid x \leftarrow u ] &= u \star \lambda x. \text{unit } t \\ [ t \mid x \leftarrow u, y \leftarrow v ] &= u \star \lambda x. v \star \lambda y. \text{unit } t \end{aligned}$$

# Parser

---

**Parser** (von engl. to parse, für: zerlegen, analysieren, (zer)gliedern) verarbeiten eine Reihe von Eingabe-Daten (z.B. einen String) entsprechend einer Grammatik (z.B. BNF-Grammatik oder Grammatik einer realen Sprache). Anwendungsgebiete: Compiler, Grammatik-Prüfer in Textverarbeitungen, o.ä.

Ein Parser heisst **eindeutig**, wenn es genau 0 oder 1 Möglichkeiten gibt, die Eingabe zu parsen. Gibt es mehr Möglichkeiten, so heisst er **mehrdeutig**.





# Ein sequenzierender Parser (Parser und Beispiele)

---

Der Parser selbst sieht so aus:

*twoItems*                    :: *M* (*Char*, *Char*)  
*twoItems*                    = *item* ★ *λa. item* ★ *λb. unit* (*a*, *b*)

Beispiele:

*twoItems* "m"                = []  
*twoItems* "Monade"        = [(['M', 'o'), "nade"]]

Dieser Parser ist auch eindeutig.

# Ein alternierender Parser (Monadendefinition)

---

Für einen Parser, der immer mehrere Alternativen liefert, definiert man:

$$\begin{aligned} \text{unit} &:: a \rightarrow M \\ \text{unit } a \ x &= [(a, x)] \end{aligned}$$

$$\begin{aligned} (\star) &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \\ (m \star k) \ x &= [(b, z) \mid (a, y) \leftarrow m \ x, (b, z) \leftarrow k \ a \ y] \end{aligned}$$

$$\begin{aligned} \text{zero} &:: M a \\ \text{zero} &= [] \end{aligned}$$

$$\begin{aligned} (\oplus) &:: M a \rightarrow M a \rightarrow M a \\ (m \oplus n) \ x &= m \ x \ ++ \ n \ x \end{aligned}$$

# Ein alternierender Parser (Parser und Beispiele)

---

Dies ist dann der zugehörige Parser:

$$\begin{aligned} \text{oneOrTwoItems} &:: M \text{ String} \\ \text{oneOrTwoItems} &= (\text{item} \star \lambda a. \text{unit } [a]) \oplus \\ & \quad (\text{item} \star \lambda a. \text{item} \star \lambda b. \text{unit } [a, b]) \end{aligned}$$

Beispiele:

$$\begin{aligned} \text{oneOrTwoItems} \text{ ""} &= [] \\ \text{oneOrTwoItems} \text{ "M"} &= [(\text{"M"}, \text{""})] \\ \text{oneOrTwoItems} \text{ "Monade"} &= [(\text{"M"}, \text{"onade"}), (\text{"Mo"}, \text{"nade"})] \end{aligned}$$

Dieser Parser ist nicht eindeutig.