Wie man das Semikolon überlädt

Code-Strukturierung und -Wiederverwendung durch Monaden

Bertram Felgenhauer

12.06.2009 / HaL4

Gliederung

Ein einfacher Termevaluator

Datentyp für Terme (`Term.hs`)

```
data Term = Const Integer
          | Binary Op Term Term
data Op = Add | Sub | Div | Mul
```

Evaluator (`eval1.hs`)

```
eval :: Term → Integer
eval (Const n) = ...
eval (Binary op a b) = ...

evalOp :: Op → Integer → Integer → Integer
evalOp Add a b = ...
```

Ein einfacher Termevaluator

Datentyp für Terme (`Term.hs`)

```
data Term = Const Integer
          | Binary Op Term Term
data Op = Add | Sub | Div | Mul
```

Evaluator (`eval1.hs`)

```
eval :: Term → Integer
eval (Const n) = ...
eval (Binary op a b) = ...

evalOp :: Op → Integer → Integer → Integer
evalOp Add a b = ...
```

Termevaluator mit Fehlerbehandlung

Ergebnistyp

$$\textbf{data}\ \mathit{MayFail}\ a = \mathit{Result}\ a \mid \mathit{Error}\ \mathit{Cause}$$

$$\textbf{data}\ \mathit{Cause} = \mathit{DivisionByZero}$$

Evaluator (`eval2.hs`)

$$\mathit{evalOp} :: \mathit{Op} \to \mathit{Integer} \to \mathit{Integer} \to \mathit{MayFail}\ \mathit{Integer}$$

$$\ldots$$

$$\mathit{evalOp}\ \mathit{Div}\ a\ 0 = \mathit{Error}\ \mathit{DivisionByZero}$$

$$\ldots$$

$$\mathit{eval} :: \mathit{Term} \to \mathit{MayFail}\ \mathit{Integer}$$

$$\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \textbf{case}\ (\mathit{eval}\ a)\ \textbf{of}$$

$$\ldots$$

Termevaluator mit Fehlerbehandlung

Ergebnistyp

$$\textbf{data}\ \mathit{MayFail}\ a = \mathit{Result}\ a \mid \mathit{Error}\ \mathit{Cause}$$

$$\textbf{data}\ \mathit{Cause} = \mathit{DivisionByZero}$$

Evaluator (`eval2.hs`)

$$\mathit{evalOp} :: \mathit{Op} \to \mathit{Integer} \to \mathit{Integer} \to \mathit{MayFail}\ \mathit{Integer}$$

...

$$\mathit{evalOp}\ \mathit{Div}\ a\ 0 = \mathit{Error}\ \mathit{DivisionByZero}$$

...

$$\mathit{eval} :: \mathit{Term} \to \mathit{MayFail}\ \mathit{Integer}$$

$$\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \textbf{case}\ (\mathit{eval}\ a)\ \textbf{of}$$

...

Fehlerbehandlung

Code-Muster

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to \mathit{MayFail}\ \mathit{Integer} \\
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \mathbf{case}\ (\mathit{eval}\ a)\ \mathbf{of} \\
&\quad \mathit{Error}\ c \to \mathit{Error}\ c \\
&\quad \mathit{Result}\ a' \to \mathbf{case}\ (\mathit{eval}\ b)\ \mathbf{of} \\
&\qquad \mathit{Error}\ c \to \mathit{Error}\ c \\
&\qquad \mathit{Result}\ b' \to \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

Idee

$$
\begin{aligned}
&(\gg\!=) :: \mathit{MayFail}\ a \to (a \to \mathit{MayFail}\ b) \to \mathit{MayFail}\ b \\
&(\mathit{Error}\ c) \gg\!= f = \mathit{Error}\ c \\
&(\mathit{Result}\ r) \gg\!= f = f\ r
\end{aligned}
$$

Fehlerbehandlung

Code-Muster

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to \mathit{MayFail}\ \mathit{Integer} \\
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \mathbf{case}\ (\mathit{eval}\ a)\ \mathbf{of} \\
&\quad \mathit{Error}\ c \to \mathit{Error}\ c \\
&\quad \mathit{Result}\ a' \to \mathbf{case}\ (\mathit{eval}\ b)\ \mathbf{of} \\
&\qquad \mathit{Error}\ c \to \mathit{Error}\ c \\
&\qquad \mathit{Result}\ b' \to \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

Idee

$$
\begin{aligned}
&(\gg\!=) :: \mathit{MayFail}\ a \to (a \to \mathit{MayFail}\ b) \to \mathit{MayFail}\ b \\
&(\mathit{Error}\ c) \gg\!= f = \mathit{Error}\ c \\
&(\mathit{Result}\ r) \gg\!= f = f\ r
\end{aligned}
$$

Fehlerbehandlung

Idee (eval2a.hs)

$$(\gg\!=) :: \mathit{MayFail}\ a \to (a \to \mathit{MayFail}\ b) \to \mathit{MayFail}\ b$$
$$(\mathit{Error}\ c) \gg\!= f = \mathit{Error}\ c$$
$$(\mathit{Result}\ r) \gg\!= f = f\ r$$

- $\gg\!=$ ist Bindeglied zwischen Berechnungen
- stellt das Ergebnis der ersten Berechnung der zweiten Berechnung zur Verfügung

Fehlerbehandlung

Idee (`eval2a.hs`)

$$(\gg\!=) :: \mathit{MayFail}\ a \to (a \to \mathit{MayFail}\ b) \to \mathit{MayFail}\ b$$
$$(\mathit{Error}\ c) \gg\!= f = \mathit{Error}\ c$$
$$(\mathit{Result}\ r) \gg\!= f = f\ r$$

- $\gg\!=$ ist Bindeglied zwischen Berechnungen
- stellt das Ergebnis der ersten Berechnung der zweiten Berechnung zur Verfügung

Fehlerbehandlung

Idee (`eval2a.hs`)

$$(\gg=) :: \mathit{MayFail}\ a \to (a \to \mathit{MayFail}\ b) \to \mathit{MayFail}\ b$$
$$(\mathit{Error}\ c) \gg= f = \mathit{Error}\ c$$
$$(\mathit{Result}\ r) \gg= f = f\ r$$

- $\gg=$ ist Bindeglied zwischen Berechnungen
- stellt das Ergebnis der ersten Berechnung der zweiten Berechnung zur Verfügung

Fehlerbehandlung

Neue Version von *eval*

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \\
&\quad \mathit{eval}\ a \gg\!= (\lambda a' \rightarrow \\
&\qquad \mathit{eval}\ b \gg\!= (\lambda b' \rightarrow \\
&\qquad\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'))
\end{aligned}
$$

oder auch

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \\
&\quad \mathit{eval}\ a \gg\!= \lambda a' \rightarrow \\
&\quad \mathit{eval}\ b \gg\!= \lambda b' \rightarrow \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

Fehlerbehandlung

Neue Version von *eval*

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \\
&\quad \mathit{eval}\ a \gg\!= (\lambda a' \rightarrow \\
&\qquad \mathit{eval}\ b \gg\!= (\lambda b' \rightarrow \\
&\qquad\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'))
\end{aligned}
$$

oder auch

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \\
&\quad \mathit{eval}\ a \gg\!= \lambda a' \rightarrow \\
&\quad \mathit{eval}\ b \gg\!= \lambda b' \rightarrow \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

Termevaluator mit Zählen von Operationen

Zähler als Zustandsvariable - funktional

$$\textbf{type } \mathit{CountOps}\ a = \mathit{Int} \rightarrow (a, \mathit{Int})$$

Evaluator (`eval3.hs`)

$$\mathit{evalOp} :: \mathit{Op} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{CountOps}\ \mathit{Integer}$$
$$\mathit{evalOp}\ \mathit{Add}\ a\ b = \lambda c \rightarrow \ldots$$
$$\ldots$$
$$\mathit{eval} :: \mathit{Term} \rightarrow \mathit{CountOps}\ \mathit{Integer}$$
$$\mathit{eval}\ (\mathit{Binary}\ op\ a\ b) = \lambda c \rightarrow \ldots$$
$$\ldots$$

Termevaluator mit Zählen von Operationen

Zähler als Zustandsvariable - funktional

$$\textbf{type } \mathit{CountOps}\ a = \mathit{Int} \to (a, \mathit{Int})$$

Evaluator (`eval3.hs`)

$$\mathit{evalOp} :: \mathit{Op} \to \mathit{Integer} \to \mathit{Integer} \to \mathit{CountOps}\ \mathit{Integer}$$
$$\mathit{evalOp}\ \mathit{Add}\ a\ b = \lambda c \to \ldots$$
$$\ldots$$
$$\mathit{eval} :: \mathit{Term} \to \mathit{CountOps}\ \mathit{Integer}$$
$$\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \lambda c \to \ldots$$
$$\ldots$$

Zählen von Operationen

Code-Muster

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \lambda \mathit{cnt} \rightarrow \\
&\quad \textbf{let}\ (a', \mathit{cnt1}) = (\mathit{eval}\ a)\ \mathit{cnt} \\
&\qquad (b', \mathit{cnt2}) = (\mathit{eval}\ b)\ \mathit{cnt1} \\
&\qquad (r, \mathit{cnt3}) = (\mathit{evalOp}\ \mathit{op}\ a'\ b')\ \mathit{cnt2} \\
&\quad \textbf{in}\ (r, \mathit{cnt3})
\end{aligned}
$$

Idee

$$
\begin{aligned}
&(\gg\!=) :: \mathit{CountOps}\ a \rightarrow (a \rightarrow \mathit{CountOps}\ b) \rightarrow \mathit{CountOps}\ b \\
&x \gg\!= f = \lambda \mathit{cnt} \rightarrow \textbf{let}\ (x', \mathit{cnt}') = x\ c\ \textbf{in}\ (f\ x')\ \mathit{cnt}'
\end{aligned}
$$

- $(\gg\!=)$ ist wieder Bindeglied zwischen Berechnungen

Zählen von Operationen

Code-Muster

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \lambda \mathit{cnt} \rightarrow \\
&\quad \textbf{let}\ (a', \mathit{cnt1}) = (\mathit{eval}\ a)\ \mathit{cnt} \\
&\qquad (b', \mathit{cnt2}) = (\mathit{eval}\ b)\ \mathit{cnt1} \\
&\qquad (r, \mathit{cnt3}) = (\mathit{evalOp}\ \mathit{op}\ a'\ b')\ \mathit{cnt2} \\
&\quad \textbf{in}\ (r, \mathit{cnt3})
\end{aligned}
$$

Idee

$$
\begin{aligned}
&(\gg\!=) :: \mathit{CountOps}\ a \rightarrow (a \rightarrow \mathit{CountOps}\ b) \rightarrow \mathit{CountOps}\ b \\
&x \gg\!= f = \lambda \mathit{cnt} \rightarrow \textbf{let}\ (x', \mathit{cnt}') = x\ c\ \textbf{in}\ (f\ x')\ \mathit{cnt}'
\end{aligned}
$$

- $(\gg\!=)$ ist wieder Bindeglied zwischen Berechnungen

Zählen von Operationen

Code-Muster

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \lambda \mathit{cnt} \rightarrow \\
&\quad \textbf{let}\ (a', \mathit{cnt1}) = (\mathit{eval}\ a)\ \mathit{cnt} \\
&\qquad (b', \mathit{cnt2}) = (\mathit{eval}\ b)\ \mathit{cnt1} \\
&\qquad (r, \mathit{cnt3}) = (\mathit{evalOp}\ \mathit{op}\ a'\ b')\ \mathit{cnt2} \\
&\quad \textbf{in}\ (r, \mathit{cnt3})
\end{aligned}
$$

Idee

$$
\begin{aligned}
&(\gg\!=) :: \mathit{CountOps}\ a \rightarrow (a \rightarrow \mathit{CountOps}\ b) \rightarrow \mathit{CountOps}\ b \\
&x \gg\!= f = \lambda \mathit{cnt} \rightarrow \textbf{let}\ (x', \mathit{cnt}') = x\ c\ \textbf{in}\ (f\ x')\ \mathit{cnt}'
\end{aligned}
$$

- $(\gg\!=)$ ist wieder Bindeglied zwischen Berechnungen

Zählen von Operationen

Noch ein Muster

$$evalOp :: Op \to Integer \to Integer \to CountOps\ Integer$$
$$evalOp\ Add\ a\ b = \lambda cnt \to (a + b, cnt)$$
$$evalOp\ Sub\ a\ b = \lambda cnt \to (a - b, cnt)$$

Idee

$$return :: a \to CountOps\ a$$
$$return\ a = \lambda cnt \to (a, cnt)$$

- *return* erlaubt, Konstanten und reine funktionale Berechnungen wiederzuverwenden.
- Beispiel: $f \gg\!= \lambda f' \to return\ (f' + 1)$

Zählen von Operationen

Noch ein Muster

$$evalOp :: Op \to Integer \to Integer \to CountOps\ Integer$$
$$evalOp\ Add\ a\ b = \lambda cnt \to (a + b, cnt)$$
$$evalOp\ Sub\ a\ b = \lambda cnt \to (a - b, cnt)$$

Idee

$$return :: a \to CountOps\ a$$
$$return\ a = \lambda cnt \to (a, cnt)$$

- *return* erlaubt, Konstanten und reine funktionale Berechnungen wiederzuverwenden.
- Beispiel: $f \gg\!= \lambda f' \to return\ (f' + 1)$

Zählen von Operationen

Noch ein Muster

$$evalOp :: Op \to Integer \to Integer \to CountOps\ Integer$$
$$evalOp\ Add\ a\ b = \lambda cnt \to (a + b, cnt)$$
$$evalOp\ Sub\ a\ b = \lambda cnt \to (a - b, cnt)$$

Idee

$$return :: a \to CountOps\ a$$
$$return\ a = \lambda cnt \to (a, cnt)$$

- *return* erlaubt, Konstanten und reine funktionale Berechnungen wiederzuverwenden.
- Beispiel: $f \gg\!= \lambda f' \to return\ (f' + 1)$

Zählen von Operationen

Noch ein Muster

$$evalOp :: Op \to Integer \to Integer \to CountOps\ Integer$$
$$evalOp\ Add\ a\ b = \lambda cnt \to (a + b, cnt)$$
$$evalOp\ Sub\ a\ b = \lambda cnt \to (a - b, cnt)$$

Idee

$$return :: a \to CountOps\ a$$
$$return\ a = \lambda cnt \to (a, cnt)$$

- *return* erlaubt, Konstanten und reine funktionale Berechnungen wiederzuverwenden.
- Beispiel: $f \gg\!= \lambda f' \to return\ (f' + 1)$

Zählen von Operationen

Idee (eval3a.hs)

$$(\gg\!=) :: \mathit{CountOps}\ a \to (a \to \mathit{CountOps}\ b) \to \mathit{CountOps}\ b$$
$$x \gg\!= f = \lambda c \to \textbf{let}\ (x', c1) = x\ c\ \textbf{in}\ (f\ x')\ c1$$

$$\mathit{return} :: a \to \mathit{CountOps}\ a$$
$$\mathit{return}\ a = \lambda cnt \to (a, cnt)$$

$$\mathit{step} :: \mathit{CountOps}\ ()$$
$$\mathit{step} = \lambda cnt \to ((), cnt + 1)$$

Zählen von Operationen

Neue Version von *eval*

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \\
&\quad \mathit{eval}\ a \gg\!= \lambda a' \rightarrow \\
&\quad \mathit{eval}\ b \gg\!= \lambda b' \rightarrow \\
&\quad \mathit{step} \gg\!= \lambda\_ \rightarrow \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

Monade

Typklasse für Monaden (vereinfacht)

class *Monad m* **where**
$(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$
$return :: a \to m\ a$

- Zum Vergleich:
  $(\gg\!=) :: MayFail\ a \to (a \to MayFail\ b) \to MayFail\ b$
- *return* entspricht $Result :: a \to MayFail\ a$
- $Error :: Cause \to MayFail\ a$ ist eine zusätzlich von *MayFail* bereitgestellte Operation.
- ähnlich für *CountOps*: $(\gg\!=)$ und *return* sind Teil der Monade; *step* ist eine zusätzliche Operation

Monade

Typklasse für Monaden (vereinfacht)

class *Monad m* **where**
$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$
$return :: a \to m\ a$

- Zum Vergleich:
  $(\gg=) :: MayFail\ a \to (a \to MayFail\ b) \to MayFail\ b$
- *return* entspricht $Result :: a \to MayFail\ a$
- $Error :: Cause \to MayFail\ a$ ist eine zusätzlich von *MayFail* bereitgestellte Operation.
- ähnlich für *CountOps*: $(\gg=)$ und *return* sind Teil der Monade; *step* ist eine zusätzliche Operation

Monade

Typklasse für Monaden (vereinfacht)

class *Monad m* **where**
$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$
$return :: a \to m\ a$

- Zum Vergleich:
  $(\gg=) :: MayFail\ a \to (a \to MayFail\ b) \to MayFail\ b$
- *return* entspricht $Result :: a \to MayFail\ a$
- $Error :: Cause \to MayFail\ a$ ist eine zusätzlich von *MayFail* bereitgestellte Operation.
- ähnlich für *CountOps*: $(\gg=)$ und *return* sind Teil der Monade; *step* ist eine zusätzliche Operation

Monade

Typklasse für Monaden (vereinfacht)

class *Monad m* **where**
$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$
$return :: a \to m\ a$

- Zum Vergleich:
  $(\gg=) :: MayFail\ a \to (a \to MayFail\ b) \to MayFail\ b$
- *return* entspricht $Result :: a \to MayFail\ a$
- $Error :: Cause \to MayFail\ a$ ist eine zusätzlich von *MayFail* bereitgestellte Operation.
- ähnlich für *CountOps*: $(\gg=)$ und *return* sind Teil der Monade; *step* ist eine zusätzliche Operation

Monade

Typklasse für Monaden (vereinfacht)

class *Monad m* **where**
$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$
$return :: a \to m\ a$

- Zum Vergleich:
  $(\gg=) :: MayFail\ a \to (a \to MayFail\ b) \to MayFail\ b$
- *return* entspricht $Result :: a \to MayFail\ a$
- $Error :: Cause \to MayFail\ a$ ist eine zusätzlich von *MayFail* bereitgestellte Operation.
- ähnlich für *CountOps*: $(\gg=)$ und *return* sind Teil der Monade; *step* ist eine zusätzliche Operation

Monade für Fehlerbehandlung

Ausprägung der Monadenklasse für *MayFail* (`eval4a.hs`)

data *MayFail a* = *Result a* | *Error String*

instance *Monad MayFail* **where**

return a = *Result a*

Result a $\gg\!=$ *f* = *f a*

Error s $\gg\!=$ *f* = *Error s*

- ersetze *Result* durch *return*
- restlicher Code wie gehabt (vergleiche `eval2a.hs`)

Monade für Fehlerbehandlung

Ausprägung der Monadenklasse für *MayFail* (`eval4a.hs`)

data *MayFail a* = *Result a* | *Error String*

instance *Monad MayFail* **where**

return a = *Result a*

Result a $\gg\!=$ *f* = *f a*

Error s $\gg\!=$ *f* = *Error s*

- ersetze *Result* durch *return*
- restlicher Code wie gehabt (vergleiche `eval2a.hs`)

Monade für Fehlerbehandlung

Ausprägung der Monadenklasse für *MayFail* (`eval4a.hs`)

data *MayFail a* = *Result a* | *Error String*

instance *Monad MayFail* **where**

return a = *Result a*

Result a $\gg=$ *f* = *f a*

Error s $\gg=$ *f* = *Error s*

- ersetze *Result* durch *return*
- restlicher Code wie gehabt (vergleiche `eval2a.hs`)

Monade für Zählen von Operationen

Ausprägung der Monadenklasse für *CountOps* (`eval4b.hs`)

$$\textbf{data}\ \mathit{CountOps}\ a = \mathit{CountOps}\ (\mathit{Int} \to (a, \mathit{Int}))$$

$$\mathit{runCountOps}\ (\mathit{CountOps}\ f) = f$$

$$\textbf{instance}\ \mathit{Monad}\ \mathit{CountOps}\ \textbf{where}$$

$$\quad \mathit{return}\ a = \mathit{CountOps}\ (\lambda \mathit{cnt} \to (a, \mathit{cnt}))$$

$$\quad \mathit{CountOps}\ x \gg\!= f = \mathit{CountOps}\ (\lambda \mathit{cnt} \to$$

$$\qquad \textbf{let}\ (x', \mathit{cnt}') = x\ \mathit{cnt}$$

$$\qquad \textbf{in}\ \mathit{runCountOps}\ (f\ x')\ \mathit{cnt})$$

- beachte: Wir brauchen einen echten Datentyp (kein Typalias) für die *instance*-Deklaration.
- vergleiche `eval3a.hs`

Monade für Zählen von Operationen

Ausprägung der Monadenklasse für *CountOps* (`eval4b.hs`)

```
data CountOps a = CountOps (Int → (a, Int))

runCountOps (CountOps f) = f

instance Monad CountOps where
  return a = CountOps (λcnt → (a, cnt))
  CountOps x >>= f = CountOps (λcnt →
    let (x', cnt') = x cnt
    in runCountOps (f x') cnt)
```

- beachte: Wir brauchen einen echten Datentyp (kein Typalias) für die *instance*-Deklaration.
- vergleiche `eval3a.hs`

Monade für Zählen von Operationen

Ausprägung der Monadenklasse für *CountOps* (`eval4b.hs`)

$$\textbf{data}\ \mathit{CountOps}\ a = \mathit{CountOps}\ (\mathit{Int} \to (a, \mathit{Int}))$$

$$\mathit{runCountOps}\ (\mathit{CountOps}\ f) = f$$

$$\textbf{instance}\ \mathit{Monad}\ \mathit{CountOps}\ \textbf{where}$$

$$\quad \mathit{return}\ a = \mathit{CountOps}\ (\lambda \mathit{cnt} \to (a, \mathit{cnt}))$$

$$\quad \mathit{CountOps}\ x \gg\!= f = \mathit{CountOps}\ (\lambda \mathit{cnt} \to$$

$$\qquad \textbf{let}\ (x', \mathit{cnt}') = x\ \mathit{cnt}$$

$$\qquad \textbf{in}\ \mathit{runCountOps}\ (f\ x')\ \mathit{cnt})$$

- beachte: Wir brauchen einen echten Datentyp (kein Typalias) für die *instance*-Deklaration.
- vergleiche `eval3a.hs`

do-Notation

- **do**-Notation ist syntaktischer Zucker für Monaden
- Folge von *Anweisungen* (Ausdrücke oder Zuweisungen)
- $\mathbf{do}\ f \implies f$
- $\mathbf{do}\ f; \ldots \implies f \gg\!= \lambda\_ \to \mathbf{do}\ \ldots$
- $\mathbf{do}\ x \leftarrow f; \ldots \implies f \gg\!= \lambda x \to \mathbf{do}\ \ldots$

Beispiel (`eval14a.hs` sowie `eval14b.hs`)

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \mathbf{do} \\
&\quad a' \leftarrow \mathit{eval}\ a \\
&\quad b' \leftarrow \mathit{eval}\ b \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

do-Notation

- **do**-Notation ist syntaktischer Zucker für Monaden
- Folge von *Anweisungen* (Ausdrücke oder Zuweisungen)
- $\mathbf{do}\ f \implies f$
- $\mathbf{do}\ f; \ldots \implies f \gg\!= \lambda\_ \to \mathbf{do}\ \ldots$
- $\mathbf{do}\ x \leftarrow f; \ldots \implies f \gg\!= \lambda x \to \mathbf{do}\ \ldots$

Beispiel (`eval14a.hs` sowie `eval14b.hs`)

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \mathbf{do} \\
&\quad a' \leftarrow \mathit{eval}\ a \\
&\quad b' \leftarrow \mathit{eval}\ b \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

do-Notation

- **do**-Notation ist syntaktischer Zucker für Monaden
- Folge von *Anweisungen* (Ausdrücke oder Zuweisungen)
- **do** $f \implies f$
- **do** $f; \ldots \implies f \gg\!= \lambda\_ \to$ **do** $\ldots$
- **do** $x \leftarrow f; \ldots \implies f \gg\!= \lambda x \to$ **do** $\ldots$

Beispiel (`eval4a.hs` sowie `eval4b.hs`)

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \mathbf{do} \\
&\quad a' \leftarrow \mathit{eval}\ a \\
&\quad b' \leftarrow \mathit{eval}\ b \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

do-Notation

- **do**-Notation ist syntaktischer Zucker für Monaden
- Folge von *Anweisungen* (Ausdrücke oder Zuweisungen)
- **do** $f \implies f$
- **do** $f; \ldots \implies f \gg\!= \lambda\_ \to$ **do** $\ldots$
- **do** $x \leftarrow f; \ldots \implies f \gg\!= \lambda x \to$ **do** $\ldots$

Beispiel (`eval14a.hs` sowie `eval14b.hs`)

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \textbf{do} \\
&\quad a' \leftarrow \mathit{eval}\ a \\
&\quad b' \leftarrow \mathit{eval}\ b \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

do-Notation

- **do**-Notation ist syntaktischer Zucker für Monaden
- Folge von *Anweisungen* (Ausdrücke oder Zuweisungen)
- $\textbf{do}\ f \implies f$
- $\textbf{do}\ f; \ldots \implies f \gg\!= \lambda\_ \to \textbf{do}\ \ldots$
- $\textbf{do}\ x \leftarrow f; \ldots \implies f \gg\!= \lambda x \to \textbf{do}\ \ldots$

Beispiel (`eval4a.hs` sowie `eval4b.hs`)

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \textbf{do} \\
&\quad a' \leftarrow \mathit{eval}\ a \\
&\quad b' \leftarrow \mathit{eval}\ b \\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

do-Notation

- **do**-Notation ist syntaktischer Zucker für Monaden
- Folge von *Anweisungen* (Ausdrücke oder Zuweisungen)
- $\textbf{do}\ f \implies f$
- $\textbf{do}\ f;\ \ldots \implies f \gg\!= \lambda\_ \to \textbf{do}\ \ldots$
- $\textbf{do}\ x \leftarrow f;\ \ldots \implies f \gg\!= \lambda x \to \textbf{do}\ \ldots$

Beispiel (`eval4a.hs` sowie `eval4b.hs`)

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Binary}\ \mathit{op}\ a\ b) = \textbf{do}\\
&\quad a' \leftarrow \mathit{eval}\ a\\
&\quad b' \leftarrow \mathit{eval}\ b\\
&\quad \mathit{evalOp}\ \mathit{op}\ a'\ b'
\end{aligned}
$$

Termevaluator mit Protokoll der Zwischenschritte

Ergebnistyp

data *Log a* = *Log* [*LogEntry*] *a*

type *LogEntry* = *String*

Zusätzliche Operation (`eval5.hs`)

addLog :: *LogEntry* → *Log* ()

- Hinweis: $(+\!\!+) :: [a] \to [a] \to [a]$ hängt Listen aneinander

Termevaluator mit Protokoll der Zwischenschritte

Ergebnistyp

data *Log a* = *Log* [*LogEntry*] *a*

type *LogEntry* = *String*

Zusätzliche Operation (`eval5.hs`)

addLog :: *LogEntry* → *Log* ()

- Hinweis: $(+\!\!+) :: [a] \to [a] \to [a]$ hängt Listen aneinander

Termevaluator mit Protokoll der Zwischenschritte

Ergebnistyp

data *Log a* = *Log* [*LogEntry*] *a*

type *LogEntry* = *String*

Zusätzliche Operation (`eval5.hs`)

addLog :: *LogEntry* → *Log* ()

- Hinweis: $(+\!\!+) :: [a] \to [a] \to [a]$ hängt Listen aneinander

Zusammenfassung

- Monaden stellen Bindeglieder für Berechnungen zur Verfügung
- klarere Programmstruktur und wiederverwendbarerer Code
- in der Regel gibt es zusätzlich zu $(\gg=)$ und *return* weitere Operationen
- Monaden sind extrem vielfältig
- Gesehen: Fehlermonade (ähnlich *Control.Monad.Error*), Zustandsmonade (*Control.Monad.State*), *Writer*-Monade (*Control.Monad.Writer*)
- Empfehlenswert: Listenmonade, *Reader*-Monade und Monadenumformer (*Control.Monad.Trans*).

Zusammenfassung

- Monaden stellen Bindeglieder für Berechnungen zur Verfügung
- klarere Programmstruktur und wiederverwendbarerer Code
- in der Regel gibt es zusätzlich zu $(\gg=)$ und *return* weitere Operationen
- Monaden sind extrem vielfältig
- Gesehen: Fehlermonade (ähnlich *Control.Monad.Error*), Zustandsmonade (*Control.Monad.State*), *Writer*-Monade (*Control.Monad.Writer*)
- Empfehlenswert: Listenmonade, *Reader*-Monade und Monadenumformer (*Control.Monad.Trans*).

Zusammenfassung

- Monaden stellen Bindeglieder für Berechnungen zur Verfügung
- klarere Programmstruktur und wiederverwendbarerer Code
- in der Regel gibt es zusätzlich zu $(\gg=)$ und *return* weitere Operationen
- Monaden sind extrem vielfältig
- Gesehen: Fehlermonade (ähnlich *Control.Monad.Error*), Zustandsmonade (*Control.Monad.State*), *Writer*-Monade (*Control.Monad.Writer*)
- Empfehlenswert: Listenmonade, *Reader*-Monade und Monadenumformer (*Control.Monad.Trans*).

Zusammenfassung

- Monaden stellen Bindeglieder für Berechnungen zur Verfügung
- klarere Programmstruktur und wiederverwendbarerer Code
- in der Regel gibt es zusätzlich zu ($\gg=$) und *return* weitere Operationen
- Monaden sind extrem vielfältig
- Gesehen: Fehlermonade (ähnlich *Control.Monad.Error*), Zustandsmonade (*Control.Monad.State*), *Writer*-Monade (*Control.Monad.Writer*)
- Empfehlenswert: Listenmonade, *Reader*-Monade und Monadenumformer (*Control.Monad.Trans*).

Zusammenfassung

- Monaden stellen Bindeglieder für Berechnungen zur Verfügung
- klarere Programmstruktur und wiederverwendbarerer Code
- in der Regel gibt es zusätzlich zu ($\gg=$) und *return* weitere Operationen
- Monaden sind extrem vielfältig
- Gesehen: Fehlermonade (ähnlich *Control.Monad.Error*), Zustandsmonade (*Control.Monad.State*), *Writer*-Monade (*Control.Monad.Writer*)
- Empfehlenswert: Listenmonade, *Reader*-Monade und Monadenumformer (*Control.Monad.Trans*).

Zusammenfassung

- Monaden stellen Bindeglieder für Berechnungen zur Verfügung
- klarere Programmstruktur und wiederverwendbarerer Code
- in der Regel gibt es zusätzlich zu ($\gg=$) und *return* weitere Operationen
- Monaden sind extrem vielfältig
- Gesehen: Fehlermonade (ähnlich *Control.Monad.Error*), Zustandsmonade (*Control.Monad.State*), *Writer*-Monade (*Control.Monad.Writer*)
- Empfehlenswert: Listenmonade, *Reader*-Monade und Monadenumformer (*Control.Monad.Trans*).

Weiterführendes Material

- Tutorials: “Monads for functional programming”
  `http://preview.tinyurl.com/zhxow`
- “You could have invented monads”
  `http://preview.tinyurl.com/cr2hvx`
- Und viele weitere
  `http://preview.tinyurl.com/pcpbel`
- Verwandte Konzepte: Idiome (*Control.Applicative*) und Pfeile (*Control.Arrow*)

Weiterführendes Material

- Tutorials: “Monads for functional programming”
  `http://preview.tinyurl.com/zhxow`
- “You could have invented monads”
  `http://preview.tinyurl.com/cr2hvx`
- Und viele weitere
  `http://preview.tinyurl.com/pcpbel`
- Verwandte Konzepte: Idiome (*Control.Applicative*) und Pfeile (*Control.Arrow*)

Ende

- Simon Peyton Jones: *"I'm on record as saying we should have called [monads] warm, fuzzy things."*
- Monica Monad (von FalconNL)