Fortgeschrittene Funktionale Programmierung

9. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2015/16

Ein zusätzliches Interface zur Parser-Bibliothek

Etwas „(gar nicht so) dunkle Magie“ (MParserCore.hs):

```
import qualified ParserCore

newtype Parser a = P (ParserCore.Parser a)
unP :: Parser a → ParserCore.Parser a
unP (P p) = p

parse :: Parser a → String → a
parse = ParserCore.parse . unP

item :: Parser Char
item = P ParserCore.item
...

instance Monad Parser where
  return  =  yield
  (>>=)  =  (++>)
  fail _   =  failure
```

Ein zusätzliches Interface zur Parser-Bibliothek

Nun Verwendung von do-Blöcken möglich (nicht erzwungen), zum Beispiel:

```
term :: Parser Expr
term = do f ← factor
          char '*'
          t ← term
          return (Mul f t)
       ||| factor
```

```
factor :: Parser Expr
factor = mapP Lit nat
         ||| do char '('
                e ← expr
                char ')'
                return e
```

statt:

```
term :: Parser Expr
term = factor ++> (\f → char '*' +++ term ++> \t → yield (Mul f t))
       ||| factor

factor :: Parser Expr
factor = mapP Lit nat
         ||| char '(' +++ expr ++> \e → char ')' +++ yield e
```

Ein-/Ausgabe in Haskell, ganz einfaches Beispiel

- In „reinen“ Funktionen ist keine Interaktion mit Betriebssystem/Nutzer/… möglich.
- Es gibt jedoch eine spezielle do-Notation, die Interaktion ermöglicht, und aus der man „normale“ Funktionen aufrufen kann.

Einfaches Beispiel:

```
prod :: [Int] → Int
prod [ ]      =  1
prod (x:xs)   =  x * prod xs
```

reine Funktion

```
main = do n ← readLn
          m ← readLn
          print (prod [n..m])
```

„Hauptschleife“

Eingabe → 5
Eingabe → 8
Ausgabe → 1680

Programmablauf

Prinzipielles zu Ein-/Ausgabe in Haskell: IO-Typen und do-Notation

- Natürlich stehen auch im Kontext von IO-behafteten Berechnungen alle Features und Abstraktionsmittel von Haskell zur Verfügung, also wir definieren Funktionen mit Rekursion, verwenden Datentypen, Polymorphie, Higher-Order, …

- Ein „komplexeres“ Beispiel:

```
dialog = do putStr "Eingabe: "
            s ← getLine
            if s == "end"
               then return ()
               else do let n = read s
                       putStrLn ("Ausgabe: " ++ show (n*n))
                       dialog
```

- Was „nein, nicht, auf keinen Fall“ geht, ist aus einem IO-Wert direkt (abseits der expliziten Sequenzialisierung und Bindung in einem do-Block) den gekapselten Wert zu entnehmen.

- Neben den gesehenen Primitiven für Ein-/Ausgabe per Terminal gibt es Primitiven und Bibliotheken für File-IO, Netzwerkkommunikation, GUIs, …

Aber fangen wir mit einem „reinen“ Beispiel an

Zur Erinnerung:

```
data Expr :: * → * where
  Lit   :: Int → Expr Int
  Add   :: Expr Int → Expr Int → Expr Int
  Sub   :: Expr Int → Expr Int → Expr Int
  Mul   :: Expr Int → Expr Int → Expr Int
  Equal :: Eq t ⇒ Expr t → Expr t → Expr Bool
  Not   :: Expr Bool → Expr Bool
  And   :: Expr Bool → Expr Bool → Expr Bool
  If    :: Expr Bool → Expr t → Expr t → Expr t

eval :: Expr t → t
eval (Lit n)       = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 − eval e2
eval (Mul e1 e2) = eval e1 * eval e2
...
```

Sinnvolle Fehlerbehandlung

Angenommen, wir wollen Division hinzufügen:

data Expr :: $* \to *$ **where**

...

Div :: Expr Int $\to$ Expr Int $\to$ Expr Int

...

Nicht so toll:

`eval` :: Expr $t \to t$

...

`eval` (Div e_1 e_2) = `eval` e_1 ‘`div`‘ `eval` e_2

...

Wegen:

```
> eval (Div (Lit 1) (Lit 0))
*** Exception: divide by zero
```

Sinnvolle Fehlerbehandlung

Eine mögliche Lösung:

```
eval :: Expr t → Maybe t
eval (Lit n)       = Just n
eval (Add e1 e2) = case eval e1 of
                     Nothing → Nothing
                     Just n1  → case eval e2 of
                                  Nothing → Nothing
                                  Just n2  → Just (n1 + n2)
...
eval (Div e1 e2)  = case eval e1 of
                     Nothing → Nothing
                     Just n1  → case eval e2 of
                                  Nothing → Nothing
                                  Just n2  → if n2 == 0
                                             then Nothing
                                             else Just (n1 `div` n2)
...
```

Sinnvolle Fehlerbehandlung

Dann:

```
> eval (Div (Lit 1) (Lit 0))
Nothing
> eval (Add (Div (Lit 1) (Mul (Lit 0) (Lit 3))) (Lit 2))
Nothing
```

Darüber hinaus jetzt möglich:

```
data Expr :: * → * where
    ...
  TryElse :: Expr t → Expr t → Expr t

eval :: Expr t → Maybe t
...
eval (TryElse e1 e2) = case eval e1 of
                         Nothing → eval e2
                         Just t  → Just t
```

Sinnvolle Fehlerbehandlung

Darüber hinaus jetzt möglich:

```
data Expr :: * → * where
    ...
  TryElse :: Expr t → Expr t → Expr t

eval :: Expr t → Maybe t
...
eval (TryElse e1 e2) = case eval e1 of
                         Nothing → eval e2
                         Just t  → Just t
```

Und dann:

```
> eval (Div (Lit 12) (TryElse (Add (Div (Lit 1) (Mul (Lit 0) (Lit 3)))
                                   (Lit 2))
                              (Lit 3)))
Just 4
```

„Plumbing“

Aber jetzt Codestruktur nicht so toll:

```
eval :: Expr t → Maybe t
eval (Lit n)       = Just n
eval (Add e1 e2) = case eval e1 of
                     Nothing → Nothing
                     Just n1  → case eval e2 of
                                  Nothing → Nothing
                                  Just n2  → Just (n1 + n2)
eval (Sub e1 e2) = case eval e1 of
                     Nothing → Nothing
                     Just n1  → case eval e2 of
                                  Nothing → Nothing
                                  Just n2  → Just (n1 − n2)
...
```

„Plumbing“

Das kriegen wir aber auch in den Griff, bzw. kürzer hin:

```
andThen :: Maybe a → (a → Maybe b) → Maybe b
andThen Nothing f = Nothing
andThen (Just n) f = f n

eval :: Expr t → Maybe t
eval (Lit n)          = Just n
eval (Add e1 e2)      = eval e1 `andThen`
                          λn1 → eval e2 `andThen`
                                  λn2 → Just (n1 + n2)
eval (Sub e1 e2)      = eval e1 `andThen`
                          λn1 → eval e2 `andThen`
                                  λn2 → Just (n1 - n2)
...
```

„Plumbing“

Das kriegen wir aber auch in den Griff, bzw. kürzer hin:

```
eval :: Expr t → Maybe t
eval (Lit n)          = Just n
eval (Add e1 e2)      = eval e1 `andThen`
                          λn1 → eval e2 `andThen`
                                  λn2 → Just (n1 + n2)
...
eval (Div e1 e2)      = eval e1 `andThen`
                          λn1 → eval e2 `andThen`
                                  λn2 → if n2 == 0
                                        then Nothing
                                        else Just (n1 `div` n2)
...
eval (TryElse e1 e2) = case eval e1 of
                         Nothing → eval e2
                         Just t  → Just t
```

Angenommen, wir wollen Variablen hinzufügen:

```
data Expr :: * -> * where
  ...
  Var :: String -> Expr Int
  ...
```

„Plumbing“:

```
eval :: Expr t -> (String -> Int) -> t
eval (Lit n)       env = n
eval (Var s)       env = env s
eval (Add e1 e2) env = eval e1 env + eval e2 env
...
```

Dann:

```
> eval (Add (Lit 3) (Var "x")) (\s -> if s == "x" then 3 else 0)
6
> eval (Add (Lit 3) (Var "y")) (\s -> if s == "x" then 3 else 0)
3
```

Refactoring analog zu vorhin:

$$\textbf{type}\ \mathsf{Env}\ a = (\mathsf{String} \to \mathsf{Int}) \to a$$

$$\texttt{andThen} :: \mathsf{Env}\ a \to (a \to \mathsf{Env}\ b) \to \mathsf{Env}\ b$$
$$\texttt{andThen}\ m\ f\ env = f\ (m\ env)\ env$$

$$\begin{aligned}
&\texttt{eval} :: \mathsf{Expr}\ t \to \mathsf{Env}\ t \\
&\texttt{eval}\ (\mathsf{Lit}\ n) &&= \texttt{const}\ n \\
&\texttt{eval}\ (\mathsf{Var}\ s) &&= \lambda env \to env\ s \\
&\texttt{eval}\ (\mathsf{Add}\ e_1\ e_2) &&= \texttt{eval}\ e_1\ \texttt{`andThen`} \\
&&&\quad \lambda n_1 \to \texttt{eval}\ e_2\ \texttt{`andThen`} \\
&&&\quad\quad \lambda n_2 \to \texttt{const}\ (n_1 + n_2)
\end{aligned}$$

...

Nun kann man sich durchaus fragen, ob sich das Refactoring in diesem Falle wirklich gelohnt hat (neben der Tatsache, dass jetzt explizit ist, an welchen Stellen *env* überhaupt relevant ist)...

Etwas bizarr, zum Zwecke der Illustration . . .

Aber stellen wir uns doch mal vor, wir möchten, dass Variablen nach erstem Auslesen auf 0 gesetzt werden, also:

```
> eval (Add (Lit 3) (Var "x")) (λs → if s == "x" then 3 else 0)
6
> eval (Add (Var "x") (Var "x")) (λs → if s == "x" then 3 else 0)
3
```

Das auf Basis von:

```
eval :: Expr t → (String → Int) → t
eval (Lit n)       env = n
eval (Var s)       env = env s
eval (Add e1 e2) env = eval e1 env + eval e2 env
...
```

machen?

Oh, das macht keinen Spaß!

Jedoch:

Von:

$$\textbf{type}\ \mathsf{Env}\ a = (\mathsf{String} \to \mathsf{Int}) \to a$$

$$\texttt{andThen} :: \mathsf{Env}\ a \to (a \to \mathsf{Env}\ b) \to \mathsf{Env}\ b$$
$$\texttt{andThen}\ m\ f\ env = f\ (m\ env)\ env$$

$$\begin{aligned}
&\texttt{eval} :: \mathsf{Expr}\ t \to \mathsf{Env}\ t \\
&\texttt{eval}\ (\mathsf{Lit}\ n) &&= \texttt{const}\ n \\
&\texttt{eval}\ (\mathsf{Var}\ s) &&= \lambda env \to env\ s \\
&\texttt{eval}\ (\mathsf{Add}\ e_1\ e_2) &&= \texttt{eval}\ e_1\ \texttt{`andThen`} \\
&&&\quad \lambda n_1 \to \texttt{eval}\ e_2\ \texttt{`andThen`} \\
&&&\quad\quad \lambda n_2 \to \texttt{const}\ (n_1 + n_2)
\end{aligned}$$

...

Jedoch:

Von:

...

zu:

$$\textbf{type}\ \text{StrangeEnv}\ a = (\text{String} \to \text{Int}) \to (a, \text{String} \to \text{Int})$$

$$\texttt{andThen} :: \text{StrangeEnv}\ a \to (a \to \text{StrangeEnv}\ b) \to \text{StrangeEnv}\ b$$
$$\texttt{andThen}\ m\ f\ env = \textbf{let}\ (a, env') = m\ env\ \textbf{in}\ f\ a\ env'$$

$$\texttt{eval} :: \text{Expr}\ t \to \text{StrangeEnv}\ t$$
$$\texttt{eval}\ (\text{Lit}\ n) = \lambda env \to (n, env)$$
$$\texttt{eval}\ (\text{Var}\ s) = \lambda env \to (env\ s, \lambda s' \to \textbf{if}\ s == s'\ \textbf{then}\ 0\ \textbf{else}\ env\ s')$$
$$\texttt{eval}\ (\text{Add}\ e_1\ e_2) = \texttt{eval}\ e_1\ \texttt{`andThen`}\ \lambda n_1 \to \texttt{eval}\ e_2\ \texttt{`andThen`}\ \lambda n_2 \to \lambda env \to (n_1 + n_2, env)$$

...

ist nicht arg so schmerzhaft.

Abstraktion!

Wir hatten jetzt:

```
andThen :: Maybe a → (a → Maybe b) → Maybe b
andThen Nothing f = Nothing
andThen (Just n) f = f n
```

und:

```
type Env a = (String → Int) → a
andThen :: Env a → (a → Env b) → Env b
andThen m f env = f (m env) env
```

und:

```
type StrangeEnv a = (String → Int) → (a, String → Int)
andThen :: StrangeEnv a → (a → StrangeEnv b) → StrangeEnv b
andThen m f env = let (a, env') = m env
                  in f a env'
```

Abstraktion!

Und wir hatten:

```
eval :: Expr t → Maybe t
eval (Lit n)       = Just n
eval (Add e1 e2) = eval e1 `andThen`
                       λn1 → eval e2 `andThen`
                                 λn2 → Just (n1 + n2)
```

bzw.

```
eval :: Expr t → Env t
eval (Lit n)       = const n
eval (Var s)       = λenv → env s
eval (Add e1 e2) = eval e1 `andThen`
                       λn1 → eval e2 `andThen`
                                 λn2 → const (n1 + n2)
```

bzw.

```
eval :: Expr t → StrangeEnv t
eval (Lit n)       = λenv → (n, env)
eval (Var s)       = λenv → (env s, λs' → if s == s' then 0 else env s')
eval (Add e1 e2) = eval e1 `andThen`
                       λn1 → eval e2 `andThen`
                                 λn2 → λenv → (n1 + n2, env)
```

Abstraktion!

Das schreit geradezu nach:

$\texttt{return} :: a \to \text{Maybe } a$
$\texttt{return } a = \text{Just } a$

und:

type Env $a = (\text{String} \to \text{Int}) \to a$

$\texttt{return} :: a \to \text{Env } a$
$\texttt{return } a = \texttt{const } a$

und:

type StrangeEnv $a = (\text{String} \to \text{Int}) \to (a, \text{String} \to \text{Int})$

$\texttt{return} :: a \to \text{StrangeEnv } a$
$\texttt{return } a = \lambda env \to (a, env)$

Abstraktion!

Dann nämlich:

```
eval :: Expr t → Maybe t
eval (Lit n)       = return n
eval (Add e1 e2) = eval e1 `andThen`
                       λn1 → eval e2 `andThen`
                                 λn2 → return (n1 + n2)
```

bzw.:

```
eval :: Expr t → Env t
eval (Lit n)       = return n
eval (Var s)       = λenv → env s
eval (Add e1 e2) = eval e1 `andThen`
                       λn1 → eval e2 `andThen`
                                 λn2 → return (n1 + n2)
```

bzw.:

```
eval :: Expr t → StrangeEnv t
eval (Lit n)       = return n
eval (Var s)       = λenv → (env s, λs' → if s == s' then 0 else env s')
eval (Add e1 e2) = eval e1 `andThen`
                       λn1 → eval e2 `andThen`
                                 λn2 → return (n1 + n2)
```

Was ist denn nun eine Monade?

Was immer wir uns wünschen, solange es (mindestens) folgendes Interface hat:

`andThen` :: M $a \to (a \to \mathrm{M}\ b) \to \mathrm{M}\ b$
`return` :: $a \to \mathrm{M}\ a$

für irgendein konkretes M.

Bzw. in Haskell eingebettet als Typkonstruktorklasse:

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

Genaugenommen auch noch:

  `fail` :: String $\to m\ a$

Was ist denn nun eine Monade?

Was immer wir uns wünschen, solange es (mindestens) folgendes Interface hat:

$\texttt{andThen} :: \mathsf{M}\ a \to (a \to \mathsf{M}\ b) \to \mathsf{M}\ b$
$\texttt{return} :: a \to \mathsf{M}\ a$

für irgendein konkretes M.

Wir haben gesehen:

$\texttt{andThen} :: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b$
$\texttt{return} :: a \to \mathsf{Maybe}\ a$

$\textbf{type}\ \mathsf{Env}\ a = (\mathsf{String} \to \mathsf{Int}) \to a$

$\texttt{andThen} :: \mathsf{Env}\ a \to (a \to \mathsf{Env}\ b) \to \mathsf{Env}\ b$
$\texttt{return} :: a \to \mathsf{Env}\ a$

$\textbf{type}\ \mathsf{StrangeEnv}\ a = (\mathsf{String} \to \mathsf{Int}) \to (a, \mathsf{String} \to \mathsf{Int})$

$\texttt{andThen} :: \mathsf{StrangeEnv}\ a \to (a \to \mathsf{StrangeEnv}\ b) \to \mathsf{StrangeEnv}\ b$
$\texttt{return} :: a \to \mathsf{StrangeEnv}\ a$

Was ist denn nun eine Monade?

Was immer wir uns wünschen, solange es (mindestens) folgendes Interface hat:

$$\texttt{andThen} :: \mathrm{M}\ a \to (a \to \mathrm{M}\ b) \to \mathrm{M}\ b$$
$$\texttt{return} :: a \to \mathrm{M}\ a$$

für irgendein konkretes M.

Außerdem fordert man noch die Gültigkeit folgender Gesetze:

$$\begin{aligned}
(\texttt{return}\ a)\ \texttt{`andThen`}\ k &= k\ a \\
m\ \texttt{`andThen`}\ \texttt{return} &= m \\
(m\ \texttt{`andThen`}\ k)\ \texttt{`andThen`}\ q &= m\ \texttt{`andThen`}\ (\lambda a \to (k\ a)\ \texttt{`andThen`}\ q)
\end{aligned}$$

(Nur leider lassen sie sich in Haskell nicht erzwingen.)

Instanzenbildung

Wenn wir mal von der Tatsache absehen, dass man von Typsynonymen „eigentlich“ nicht einfach Typklasseninstanzen bilden darf, kriegen wir:

instance Monad Maybe **where**

$$
\begin{aligned}
&\texttt{fail}\ \_ &&= \text{Nothing}\\
&\texttt{return}\ a &&= \text{Just}\ a\\
&\text{Nothing} \gg\!= f &&= \text{Nothing}\\
&\text{Just}\ n \gg\!= f &&= f\ n
\end{aligned}
$$

instance Monad Env **where**

$$
\begin{aligned}
&\texttt{return}\ a &&= \texttt{const}\ a\\
&m \gg\!= f &&= \lambda env \to f\ (m\ env)\ env
\end{aligned}
$$

instance Monad StrangeEnv **where**

$$
\begin{aligned}
&\texttt{return}\ a &&= \lambda env \to (a, env)\\
&m \gg\!= f &&= \lambda env \to \textbf{let}\ (a, env') = m\ env\\
& && \quad\ \textbf{in}\ f\ a\ env'
\end{aligned}
$$

. . . und entsprechender Einsatz:

$$
\begin{aligned}
&\texttt{eval} :: \text{Expr } t \to \text{Maybe } t \\
&\texttt{eval } (\text{Lit } n) \quad\quad = \texttt{return } n \\
&\texttt{eval } (\text{Add } e_1\ e_2) = \texttt{eval } e_1 \gg\!= \\
&\qquad\qquad\qquad\qquad \lambda n_1 \to \texttt{eval } e_2 \gg\!= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda n_2 \to \texttt{return } (n_1 + n_2) \\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{eval} :: \text{Expr } t \to \text{Env } t \\
&\texttt{eval } (\text{Lit } n) \quad\quad = \texttt{return } n \\
&\texttt{eval } (\text{Var } s) \quad\quad = \lambda env \to env\ s \\
&\texttt{eval } (\text{Add } e_1\ e_2) = \texttt{eval } e_1 \gg\!= \\
&\qquad\qquad\qquad\qquad \lambda n_1 \to \texttt{eval } e_2 \gg\!= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda n_2 \to \texttt{return } (n_1 + n_2) \\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{eval} :: \text{Expr } t \to \text{StrangeEnv } t \\
&\texttt{eval } (\text{Lit } n) \quad\quad = \texttt{return } n \\
&\texttt{eval } (\text{Var } s) \quad\quad = \lambda env \to (env\ s, \lambda s' \to \textbf{if } s == s' \textbf{ then } 0 \textbf{ else } env\ s') \\
&\texttt{eval } (\text{Add } e_1\ e_2) = \texttt{eval } e_1 \gg\!= \\
&\qquad\qquad\qquad\qquad \lambda n_1 \to \texttt{eval } e_2 \gg\!= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda n_2 \to \texttt{return } (n_1 + n_2)
\end{aligned}
$$

Lohnt der ganze Aufwand?

Was haben wir denn nun eigentlich gewonnen (neben der rein syntaktischen Ähnlichkeit verschiedener Auswertungsfunktionen)?

Nun, hätten wir unseren Ursprungsauswerter gleich so geschrieben:

$$
\begin{aligned}
&\texttt{eval} :: \text{Expr } t \to t \\
&\texttt{eval } (\text{Lit } n) \quad\quad = \texttt{return } n \\
&\texttt{eval } (\text{Add } e_1\ e_2) = \texttt{eval } e_1 \gg\!= \\
&\qquad\qquad\qquad\qquad \lambda n_1 \to \texttt{eval } e_2 \gg\!= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda n_2 \to \texttt{return } (n_1 + n_2) \\
&\ldots \\
&\texttt{eval } (\text{Not } e) \quad\quad = \texttt{eval } e \gg\!= \lambda n \to \texttt{return } (\texttt{not } n) \\
&\ldots
\end{aligned}
$$

bzw. sogar mit dem allgemeineren Typ

$$\texttt{eval} :: \text{Monad } m \Rightarrow \text{Expr } t \to m\ t$$

(aber gleicher Definition), dann ...

Lohnt der ganze Aufwand?

. . . hätten wir unsere Spracherweiterungen sehr fokussiert umsetzen können:

für Exceptions: lediglich hinzufügen (und aus m wird Maybe):

$$\begin{aligned}
\texttt{eval}\ (\text{Div}\ e_1\ e_2) \quad &= \texttt{eval}\ e_1 \gg\!= \\
&\quad \lambda n_1 \to \texttt{eval}\ e_2 \gg\!= \\
&\quad\quad \lambda n_2 \to \textbf{if}\ n_2 == 0 \\
&\quad\quad\quad \textbf{then}\ \texttt{fail}\ \texttt{"..."} \\
&\quad\quad\quad \textbf{else}\ \texttt{return}\ (n_1\ \texttt{`div`}\ n_2) \\
\texttt{eval}\ (\text{TryElse}\ e_1\ e_2) &= \textbf{case}\ \texttt{eval}\ e_1\ \textbf{of} \\
&\quad \text{Nothing} \to \texttt{eval}\ e_2 \\
&\quad \mathit{okay} \quad\ \to \mathit{okay}
\end{aligned}$$

für Variablen: lediglich hinzufügen (und aus m wird Env):

$$\texttt{eval}\ (\text{Var}\ s) = \lambda env \to env\ s$$

für „bizarre“ Variablen: lediglich hinzufügen (und aus m wird StrangeEnv):

$$\texttt{eval}\ (\text{Var}\ s) = \lambda env \to (env\ s, \lambda s' \to \textbf{if}\ s == s'\ \textbf{then}\ 0\ \textbf{else}\ env\ s')$$

Allerdings. . .

. . . mindestens zwei große „Aber“:

1. Von dem ursprünglichen Auswerter:

$$
\begin{aligned}
&\texttt{eval} :: \text{Expr } t \to t \\
&\texttt{eval } (\text{Lit } n) \quad\;\; = n \\
&\texttt{eval } (\text{Add } e_1\ e_2) = \texttt{eval } e_1 + \texttt{eval } e_2 \\
&\texttt{eval } (\text{Sub } e_1\ e_2) = \texttt{eval } e_1 - \texttt{eval } e_2 \\
&\texttt{eval } (\text{Mul } e_1\ e_2) = \texttt{eval } e_1 * \texttt{eval } e_2 \\
&\ldots
\end{aligned}
$$

umzuschwenken auf:

$$
\begin{aligned}
&\texttt{eval} :: \text{Monad } m \Rightarrow \text{Expr } t \to m\ t \\
&\texttt{eval } (\text{Lit } n) \quad\;\; = \texttt{return } n \\
&\texttt{eval } (\text{Add } e_1\ e_2) = \texttt{eval } e_1 \gg\!= \\
&\qquad\qquad\qquad\qquad \lambda n_1 \to \texttt{eval } e_2 \gg\!= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda n_2 \to \texttt{return } (n_1 + n_2) \\
&\ldots
\end{aligned}
$$

erfordert natürlich einen gewissen Aufwand.

Allerdings. . .

. . . mindestens zwei große „Aber“:

2. Wer sagt uns eigentlich, dass das Ganze auch jenseits unserer drei Anwendungsfälle „Exceptions“, „Variablen“ und „bizarre Variablen“ trägt?

   Und dass das Power-to-Weight-Ratio stimmt?

Zunächst zu 1.:

Spezielle Compilerunterstützung für „**do**-Notation“:

Aus:

```
eval :: Monad m => Expr t -> m t
eval (Lit n)      = return n
eval (Add e1 e2) = eval e1 >>=
                       \n1 -> eval e2 >>=
                                  \n2 -> return (n1 + n2)
...
```

wird:

```
eval :: Monad m => Expr t -> m t
eval (Lit n)      = return n
eval (Add e1 e2) = do n1 <- eval e1
                      n2 <- eval e2
                      return (n1 + n2)
...
```

Außerdem im Prinzip automatische Unterstützung (zum Übergang von etwa `eval (Add e1 e2) = eval e1 + eval e2`) möglich.

Zunächst zu 1.:

Realisierung der „**do**-Notation“: einfache Syntaxtransformation

$$
\begin{array}{lcl}
\textbf{do}\ e & \rightsquigarrow & e \\
\textbf{do}\ p \leftarrow e & \rightsquigarrow & \textbf{let}\ \texttt{ok}\ p = \textbf{do}\ \textit{stmts} \\
\quad \textit{stmts} & & \qquad \texttt{ok}\ \_ = \texttt{fail}\ \texttt{"..."} \\
& & \textbf{in}\ e \gg\!= \texttt{ok} \\
\textbf{do let}\ \textit{decls} & \rightsquigarrow & \textbf{let}\ \textit{decls} \\
\quad \textit{stmts} & & \textbf{in do}\ \textit{stmts} \\
\textbf{do}\ e & \rightsquigarrow & e \gg\!= \lambda\_ \rightarrow \textbf{do}\ \textit{stmts} \\
\quad \textit{stmts} & &
\end{array}
$$

- wirklich nur syntaktischer Zucker (wie list comprehensions)
- `return` hat nichts mit C (oder so) zu tun: ganz normale Funktion; „springt“ nicht; ist nicht immer am Blockende; ...
- nicht jede monadische Berechnung innerhalb **do**-Block
- manchmal nötig, **do**-Blöcke zu schachteln (da exakt, und nur, obige Regeln verwendet)

Nun zu 2.:

2. Wer sagt uns eigentlich, dass das Ganze auch jenseits unserer drei Anwendungsfälle „Exceptions“, „Variablen“ und „bizarre Variablen“ trägt?

   Und dass das Power-to-Weight-Ratio stimmt?

- Beweis durch Autorität: . . . sonst hätte man sich wohl nicht die Mühe der Sonderbehandlung im Compiler gemacht.
- Beweis durch Masse: $\rightarrow$ Hoogle, insbesondere die Vielzahl an Funktionen, die polymorph über Monaden sind
- Beweis durch Ausprobieren: . . .