Eine Führung durch Monaden für junge Anwender (draft)

Sven Eric Panitz
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail: panitz@informatik.uni-frankfurt.de

17. Januar 1996

Zusammenfassung

Monadisches Programmieren ist für viele Programmierer, die sich auf das funktionale Programmieren willig eingelassen haben, weiterhin ein rotes Tuch. Aus diesen Grunde gibt es eine Reihe von verschiedenen Papieren, die auf unterschiedliche Weise und mit unterschiedlichen Motivationen, den Leser an das Problem heranführen wollen [Wad90, Wad92].

Da das Haskell-Komitee dieser Tage das monadische I/O-Konzept nun zum offiziellen Standard in Haskell 1.3 erkiest, kommt auch eine Vorlesung über funktionales Programmieren nicht umhin, dieses Thema zu behandeln. Daher versuche ich mich hier an einer weiteren Einführung in Monaden.

1 Konstruktorklassen

Wir haben bereits gesehen, wie man in Haskell 1.3 eine erweiterte Möglichkeit hat, Klassen von Typen zu definieren, die sich dadurch auszeichnen, daß auf ihnen bestimmte Funktionen überladen existieren [Jon95].

1.1 Die Funktorklasse

Als Beispiel diente uns die Überladung der Funktion `map` auf mehrere Typen. Hierzu hatten wir auf Übungsblatt 5 für verschiedene Typen eine Funktion `map` definiert:

```
> data Tree a = TNil
>              |TCons (Tree a) a (Tree a)

> data Foo a = Eins a (Foo a) |List a |String

> maptree f TNil = TNil
> maptree f (TCons t1 a t2)
>   = TCons (maptree f t1) (f a) (maptree f t2)
```

```
> mapfoo f String = String
> mapfoo f (List a) = List (f a)
> mapfoo f (Eins a foo)  = Eins (f a) (mapfoo f foo)

> maplist f [] = []
> maplist f (x:xs) = (f x):(map f xs)
```

Ziel war es, die drei Funktionen als eine Funktion darzustellen. Hierzu konnten wir eine Konstruktorklasse definieren.

```
> class Functor c where
>   mapp :: (a -> b) -> (c a -> c b)
```

Die Variable `c` steht in dieser Definition für einen Typkonstruktor eines algebraischen Typen. Wir können für diese Variable nun Instanzen von `mapp` definieren. Wir setzen z.B. `c` als `Tree`. Dann brauchen wir eine Funktion `mapp` mit dem Typen: `(a -> b) -> (Tree a -> Tree b)`. Eine solche Funktion haben wir oben gerade definiert. Wir benutzen sie als Instanz:

```
> instance Functor Tree where mapp = maptree
```

Ebenso können wir für Listen und `Foo` verfahren:

```
> instance Functor Foo where mapp = mapfoo

> instance Functor [] where mapp = map
```

1.2 Die Monadenklasse

Wir wollen uns jetzt eine weitere Konstruktorklasse definieren. Sie wird Typen zusammenfassen, auf denen bei manchen Programmierern gern benutzte Funktionen definiert sind. Die Monadenklasse braucht zunächst als Vorbedingung, daß die Typen, die in ihr enthalten sind, auch in der Funktorklasse sind; sprich: daß `mapp` auf ihnen definiert ist.

```
> class Functor m => Monad m where
>   result :: a -> m a
>   join   :: m (m a) -> m a
>   bind   :: m a -> (a -> m b) -> m b
>
>   join x = bind x id
>   x `bind` f = join (mapp f x)
```

Die Klasse `Monad` sammelt also alle algebraischen Typen, auf denen Instanzen der Funktionen `result`, `bind` und `join` definiert sind. Hierbei brauch aber von `bind` und `join` nur eine Funktion explizit definiert sein. Die andere ist dann über die *Default*-Definition instanziiert. Dieses ist ähnlich, wie in der Gleichheitsklasse `Eq` nur die Gleichheit definiert zu sein braucht, und man die Ungleichheit über eine *Default*-Gleichhung erhält.

Bisher kennen wir nur die Typsignaturen der Funktionen in der Monadenklasse. Betrachten wir die Funktionen im einzelnen:

- `result`:
  Dieses ist offenbar eine Funktion, die eine Eingabe nimmt und diese für den algebraischen Typ zu einer Art Element macht. Aus Listen kann man sich also etwa vorstellen, daß ein Element in eine Liste eingebaut wird. Da es keine weitere Eingabeliste gibt, wird es wohl sinnvoll sein, hier eine einelementige Liste zu definieren.

- `bind`:
  Diese Funktion sieht ein bißchen wie `map` aus. Es wird ein Ausdruck eines algebraischen Typs genommen, dann wird eine Funktion genommen, die die Elemente des ersten Ausdrucks verarbeitet und zu eventuell einer anderen Art von Elementen des ursprünglichen Typs machen. Ergebnis ist dann die eventuell andere Instanz dieses Typens.

- `join`:
  Diese Funktion ist über die *Default*-Gleichhung definiert, wenn wir `bind` kennen. Daher wollen wir sie jetzt nicht weiter betrachten.

Es ist wohl an der Zeit, Beispiele für Instanzen der Monadenklasse zu geben. Ein der Vorstellung wohl am nächsten liegendes Beispiel sind die Listen.

```
> instance Monad [] where
>    result a = [a]
>    [] `bind` k = []
>    (x:xs) `bind` k = (k x) ++ (xs `bind` k)
```

Man vergewissere sich, daß diese Instanzdefinition, den verlangten Typen entspricht. Wir haben uns Die Monadenfunktionen auf Listen tatsächlich so definiert, wie uns die Typsignatur nahegelegt hat. `result` nimmt ein Element und baut mit ihm eine einelementige Liste. `bind` nimmt eine Liste, baut aus jedem Listenelement eine neue Liste, die dann konkateniert werden.

Wir haben `bind` in Infixschreibweise angewendet. Dieses wird später sich als recht sinnvoll erweisen.

Was zeichnet aber nun eine Monade allgemein aus? Warum haben wir diese spezielle Konstruktorklasse eingeführt?

Die Funktionen `result` und `bind` auf einem algebraischen Datentypen qualifizieren sich für Monaden, wenn sie drei algebraische Gesetze erfüllen:

$$\begin{aligned} \texttt{(result a) `bind` k} &= \texttt{k a} \\ \texttt{m `bind` result} &= \texttt{m} \\ \texttt{m `bind` (\textbackslash a -> (k a) `bind` h)} &= \texttt{(m `bind` (\textbackslash a -> (k a)) `bind` h} \end{aligned}$$

Man vergewissere sich als Übung, daß diese drei Gesetze für die Monadenfunktionen auf Listen gelten. Ohne daß wir das syntaktisch sicherstellen können, wollen wir nur Instanzen von der Konstruktorklasse `Monad` definieren, die diesen drei Gestzen genügen.

2 Parser

Bisher ist uns noch nicht ganz klar geworden, welchen praktischen Nutzen wir aus der Monadenklasse ziehen können. Deshalb wollen wir jetzt versuchen, für eine uns wohlbekannte Anwendung eine Lösung zu schreiben, in der wir Instanzen der Monadenklasse benutzen. Wir hatten uns auf dem letzten Übungsblatt mit Parsern beschäftigt. Wir werden feststellen, daß wir dort schon fast monadisch programmiert haben.

Wir hatten uns bei der Entwicklung eines Parsers an [FL89] orientiert. Ein Parser war dabei eine Funktion von einer Tokenliste, auf eine Liste von Parsergebnissen. Ein Parsergebnis stellte ein Paar von semantischen Ergebnis des Parsens und der Restliste, die noch nicht geparst werden konnte, dar.

Machen wir zunächst aus dem Typ eines Parsers einen algebraischen Typ:

```
> data Parser a = P (String -> [(a,String)])
```

Ein Parser ist jetzt also ein algebraischer Typ, der über seinen Ergebnistyp polymorph ist.[1] Daß in diesem Typ eine Funktion versteckt ist, braucht uns zunächst nicht zu beunruhigen.

Beginnen wir die Implementierung eines monadischen Parsers mit der Definition einer Funktion `mapp` für Parser. Parser sind polymorph über ihren Ergebnistypen. Also soll `mapp` das Ergebnis eines Parses ändern, ohne den Pars zu ändern. Wir können dieses wie folgt definieren:

```
> instance Functor Parser where
>     mapp f (P p)
>       = P (\x -> (mapp (\(a,b)->(f a,b)) (p x)))
```

Wer sich genau entsinnt, den mag diese Funktion vertraut vorkommen. Es entspricht nämlich in etwa der Funktion `sem` in unseren bisherigen Parsern.

```
sem p f = \xs -> (map \(x,y)->(f x,y) (p xs))
```

Der einzige Unterschied ist, daß jetzt unsere Parser mit einem Konstruktor `P` verpackt sind.[2] Damals hatten wir die Funktion allerdings etwas gezuckert mit einer Listcomprehension geschrieben:

[1] Die Tokenliste ist in diesem Beispiel der Einfachheit halber als `String` definiert.

[2] Inwieweit das notwendig war, muß ich noch einmal abchecken. Lieber hätte ich statt `data Parser a = P ...` nur `type Parser a = ...` definiert. Dann habe ich aber die Instanzdefinition für die Konstruktorklasse `Monad` nicht hinbekommen. Ob das aber nur ein Fehler mit Gofer war, oder konzeptuell nicht geht, vermag ich noch nicht zu sagen.

```
sem p f = \xs -> [(f x,y) | (x,y) <- p xs]
```

Jetzt werden wir kühn und wollen `Parser` auch zu Instanzen von der Monadenklasse machen. Hierzu müssen wir uns überlegen, wie wir sinnvoll die Funktionen `result` und `bind` für Parser instaziieren können.

- `result`:
  Hier bleibt uns eigentlich kaum eine Wahl. Wir sollen eine Funktion schreiben, die ihr Argument nimmt und einen Parser als Ergebnis hat, der genau dieses Argument als Ergebnis ausgibt. Da wir keine weiteren Anweisungen haben, wie denn dieser Parser aussehen soll, ist nur der primitivste Parser, den man sich vorstellen kann, sinnvoll. Dieses ist der Parser, der einer ϵ-Regel entspricht, der also nur das leere Wort von der Tokenliste verbraucht.

- `bind`:
  Hier erhalten wir als erstes Argument einen Parser. Den sollen wir verbinden mit seinem zweiten Argument. Dieses ist nicht direkt selber ein Parser, aber eine Funktion, die vom ersten Parser das Ergebnis übernimmt und es benutzt einen zweiten Parser zu konstruieren. Das Ergebnis soll offensichtlich ein Parser sein, der die Parsgänge und Ergebnisse der beiden Parser sequentiell kombiniert.

Wir können die Instanz jetzt wie folgt bilden:

```
> instance Monad Parser where
>    result a = P (\x->[(a,x)])
>    (P p) `bind` k
>      = P (\x->[(b,z)|
>                  (a,y)<-p x,
>                  (b,z)<- let (P q) = k a in q y])
```

Man vergewissere sich zunächst davon, daß die Gesetze auf Monaden mit diesen Definitionen erfüllt sind. `result` ist jetzt tatsächlich so definiert, wie oben angedeutet. Wer sich erinnern mag, erkennt den ϵ-Parser aus unseren bisherigen Parsern wieder: `epsilon xs = [(id,xs)]`.

`bind` erinnert hingegen etwas an `seq` in unseren bisherigen Parsern:

```
seq p q
 = \xs -> [ ((erg1,erg2),xs2)|
                (erg1,xs1) <- p xs,
                (erg2,xs2) <- q xs1 ]
```

Der Unterschied liegt lediglich darin, daß wir nicht zwei Parser sequenziell miteinander verbinden; das zweite Argument ist nicht nur ein Parser, sondern hat als Funktion auch gleich die Information, auf welche Weise die beiden Ergebnisse der nacheinander ausgeführten Parsgänge zu verknüpfen sind. Es finden sich im Prinzip die Funktionen `seq` und `sem` in der Funktion `bind` vereinigt.

Ähnlich wie wir uns für `seq` die Arbeit sehr vereinfachen konnten, indem wir für die Funktion einen Infixoperator eingeführt haben, können wir auch jetzt verfahren:

```
> infixr 8 ==>

> (==>) :: Monad c =>  c a -> (a -> c b) -> c b
> (==>) = bind
```

Damit hätten wir fast alle Elemente des Parsers, die wir in der letzten Woche kennengelernt haben, monadisch ausgedrückt. Lediglich die Alternativkombination zweier Parser haben wir noch nicht betrachtet. Diese kann aber fast genauso gehandhabt werden, wie zuvor:

```
> alt (P p) (P q) = P (\xs-> (p xs)++(q xs))
```

Es hat uns die monadische Betrachtung unseres Parsers im Prinzip noch keine Vorteile gebracht, waren wir doch mit den bisherigen Parsern auch recht zufrieden. Was wir allerdings gesehen haben, ist, daß das monadische Programmieren gar nicht so sehr Neuland ist, wie zunächst erwartet. Es funktioniert etwa nach den gleichen Prinzipien, wie die uns bekannten Parserkombinatoren. Dabei war die hier vorgestellte Monade schon recht kompliziert.

In den nächsten Abschnitten werden wir aber weitere Lösungen von Programmierproblemen kennenlermnen, in denen uns der allgemeine Rahmen von Monaden ein große Hilfe sein kann.

3 Erweitern von Programmen

In diesem Abschnitt wollen wir zeigen, wie man in der Programmierung flexibler werden kann durch die Benutzung von Monaden. Das vorgestellte Programm stammt ursprünglich aus [Wad95] Hierzu schreiben wir ein Minimalprogramm, das bestimmte arithmetische Ausdrücke auswertet:

```
> data Term = Con Int | Div Term Term

> eval (Con a) = a
> eval (Div t u) = (eval t) / (eval u)
```

Wir wollen diesen Auswerter monadisch schreiben. Hierzu werden wir das Ergebnis der Auswertung in einen Typen packen, der in der Monadenklasse ist. Je nachdem, welchen zusätzlichen Ansprüche unser Auswerter dann noch genügen soll, kann dieses eine andere Monade sein:

```
> val (Con a) = result a
> val (Div t u) = val t          ==> (\a ->
>                 val u          ==> (\b ->
>                 rechneresult a b          ))
>
```

```
  rechneresult a b = result (a / b)
```

Die Funktionen `eval` und `val` sind nicht dieselben. Sie haben unterschiedliche Typen:

```
eval :: Term -> Int
val :: Monad a => Term -> a Int
```

Das bedeutet, daß erst im Kontext hervorgehen kann, welchen Typ ein Ausdruck der Form `val (Con 6)` hat:

```
? val (Con 6)

ERROR: Unresolved overloading
*** type        : Monad a => a Int
*** translation : val (Con 6)

? :t  (head (val (Con 6)))
head (val (Con 6)) :: Int
```

Im zweiten Aufruf ist deutlich geworden, daß wir den Auswerter als Listenmonade benutzt haben. Vielleicht wollen wir das Ergebnis der Auswertung in überhaupt keine spezielle Monade "verpacken". Hierzu können wir uns eine *Nullmonade* definieren:

```
> data I a = I a

> instance Functor I where mapp f (I a) = I (f a)
>
> instance Monad I where
>    result a = I a
>    m `bind` k = k a where (I a) = m
>
> i (I a) = a
```

In dieser Monade wird lediglich ein eindeutiger Konstruktor vor das Ergebnis geschrieben.[3] Der Auswerter funktioniert ebenso in dieser Monade:

```
? i  (val (Div (Con 6) (Con 3)))
2
(20 reductions, 37 cells)
```

Jetzt wollen wir den Auswerter um eine explizite Fehlerbehandlung erweitern. Ein möglicher Fehler der Auswertung ist die Division durch 0. Wir definieren uns einen algebraischen Datentypen, der unterscheidet, ob ein Fehler auftrat oder ein Ergebnis berechnet wurde. Diesen Datentypen machen wir dann in naheliegender Weise zu einer Instanz der Monadenklasse:

[3] Wenn Gofer es zulassen würde, Instanzen der Monadenklasse von `type I a = a` zuzulassen, so bräuchten wir nichteinmal den Konstruktor `I`.

```
> data Error a = Error Ausnahme | Return a
> type Ausnahme = String
>
> instance Functor Error where
>    mapp f (Error s) = Error s
>    mapp f (Return a) = Return (f a)
>
> instance Monad Error where
>     result a = Return a
>     join (Return a) = a
>     join (Error s) = Error s
>
> erg (Return a) = Return a
> erg          s = s
```

Hier haben wir zum ersten Mal die Monade nicht über die Funktion `bind`, sondern über die Funktion `join` definiert. `bind` erhält dann seine Definition durch eine Defaultgleichung. Wir brauchen jetzt nur noch die Funktion `rechne` so abzuändern, daß sie auch Ergebnisse vom Typ `(Error a)` berechnet.

```
  rechneresult a b
   |b==0 = Error "Division durch null\n"
   |otherwise = result (a/b)
```

Damit haben wir den Auswerter auf einfache Art um eine Fehlerbehandlung erweitert. Wir haben die Berechnungsfunktion abgeändert und eine neue Instanz der Monadenklasse definiert:

```
? erg (val (Div (Div (Con 6) (Con 0)) (Con 1)))
Error "Division durch null\n"
(45 reductions, 127 cells)
```

In einem weiteren Schritt wollen wir das Programm jetzt so abändern, daß wir bei der Berechnung des Ergebnisses im Auswerter noch eine statistische Information mitberechnen, nämlich die Schritte, die die erechnung benötigt (ähnlich der Reduktionsanzahl, die Gofer angibt). Wir definieren hierzu einen Datentypen, der sowohl das Ergebnis als auch die bereits getätigten Berechnungsschritte enthält:

```
> data StateM a = S (State -> (a,State))
> type State = Int
>
> instance Functor StateM where
>     mapp f (S g)
>       = S (\s -> (\(a,b)->(f a,b)) (g s))

> instance Monad StateM where
>     result a =S (\x -> (a,x))
>     (S m) `bind` k =S (\x -> let (a,y) = m x     in
```

```
>                               let (b,z) = app (k a) y    in
>                               (b,z))
>
> app (S s) a = s a
```

Diese Monade erinnert stark an die Parsermonade im letzten Abschnitt. Damals war der State die noch abzuarbeitende Tokenliste. Die Berechnung eines Ergebnisses soll jetzt noch das Hochsetzen des Zählers um 1 beinhalten:

```
> rechneresult a b = tick ==> (\() -> result (a/b))
> tick = S ( \x -> ((),x+1))
```

Damit erhalten wir den erwünschten Auswerter mit Zählfunktion.

```
?  app (val (Div (Div (Con 6) (Con 3)) (Con 1))) 0
(2,2)
(81 reductions, 195 cells)
```

Die letzte Variation des Auswerters macht besonders deutlich den sequentiellen Aspekt der Funktion `bind`. Wie schon bei den Parsers ist die Funktion fast imperativ zu lesen als: werte erst den Ausdruck des ersten Arguments aus, reiche das Ergebnis an den Ausdruck des zweiten Arguments weiter und werte dann diesen aus.

4 Monadisches I/O

Wir haben den letzten Abschnitt damit geschlossen, auf den starken sequentiellen Charakter der Funktion `bind` hinzuweisen. Dieses ermöglicht und ein I/O-Konzept auf dem Prinzip der Monaden aufzubauen. Dieses wurde vorgestellt in [PJW93] Wir erinnern uns, daß wir für das I/O in reinen funktionalen Sprachen zwei Probleme hatten:

- **Sequenzialisierung**: es ist kaum für einen Programmierer sicherzustellen, in welcher Reihenfolge die Ausdrücke ausgeweret werden.
- **referenzielle Transparenz**: wie ist die Semantik von Aufrufen an das Betriebssystem referentiell Transparent zu definieren? Das Ergebnis des Aufrufes hängt ja von den Zustand der Systemumgebung ab.

Für das Problem der Sequentialisierung haben wir mit den Monaden eine adequate Lösung gefunden. Wir müssen nun nur noch eine spezielle I/O-Monade entwerfen, die dafür sorgt, daß wir ein referentiell transparentes I/O-Konzept bekommen. Die Lösung hierzu ist, daß der gesammte Systemzustand implizit ein Argument der I/O-Funktionen ist. Damit haben wir dann die referentielle Transparenz auch durch *enviroment passing* für die I/O Funktionen gewährleistet.

Der Datentyp `(IO a)` soll uns die IO-Monade definieren:

```
type IO a = World -> IORes a
data IORes a = MkIORes a World
```

Der Datentyp `World` spiegelt dabei den Systemzustand wieder. Die Monadeninstanz ist dann definiert als:

```
result a w    = MkIORes a w
m `bind` k    = \w -> case (m w) of
                           MkIORes a w' -> k a w'
```

Der Typ `World` ist ein abstrakter Typ mit genau einer auf ihm definierten Funktion, der Funktion `ccall`. Diese Funktion ist das einzige Sprachkonstrukt, daß Haskell zum Implementieren des monadischen I/O hinzugefügt wurde. Es erlaubt eine beliebige C-Funktion aufzurufen, welche dann die Kommunikation mit der Umgebung betreibt. Mit Hilfe dieser Funktion werden die I/O-Funktionen definiert. Betrachten wir hierzu ein Beispiel:

```
putcIO a =\w ->
  case a of
    MkChar a# ->
      case (ccall# putchar a# w) of
       MkIORes# n# w' -> MkIORes () w'
```

Jetzt läßt sich in ähnlicher Weise wie mit Continuation-basierten I/O unser kleiner Editor schreiben. Wir geben einen Ausschnitt des Editors, der vom Continuation-basierten I/O ins monadische I/O, wie es der ghc derzeit zur Verfügung stellt, übersetzt wurde:

```
main = putStr "ZeilenEditor\n"    >>
       newcommand []

newcommand file
 = readLine                       >>= \eingabe ->
   mainloop (eingabe,file)

readLine
  = isEOF                         >>= \eof ->
       if eof then return [] else getChar
                                  >>= \c   ->
       if c `elem` ['\n'] then return [] else readLine
                                  >>= \cs  ->
       return (c:cs)

mainloop ((':':'l':' ':dateiname), buffer)
 = try (readFile dateiname)       >>= \res ->
   case res of
    (Left _)
       -> putStr("Datei "++dateiname
                  ++" konnte nicht gelesen werden!\n") >>
          newcommand buffer
    (Right file) ->  newcommand (lines file)

mainloop ((':':'s':' ':dateiname), buffer)
 = try (writeFile dateiname (unlines buffer)) >>= \res ->
   (case res of
    (Left _)
      -> putStr("Datei "++dateiname
                 ++" konnte nicht geschrieben werden!\n")
    (Right _)
      ->  putStr (dateiname++" wurde geschrieben!\n")) >>
```

```
   newcommand buffer

mainloop ([':','d'],buffer)
 = putStr (unlines (number buffer))                    >>
   newcommand buffer
    where
     number buffer = [(show n)++". "++l|(n,l)<-zip [1..] buffer ]
```

Die Funktion `bind` ist in der derzeitigen Implementierung zur Infixnotation `>>=` geworden. Hierbei sehen wir, daß dieses immer gefolgt wird von einer Funktion `\res ->`. Dieses entspricht den erwarteten Typ von `bind`. Wir sehen in diesem Programm aber auch einen weiteren Infixoperator: `>>` Dieser wird nicht von einer Funktion, die eine IO-Monade als Ergebnis hat, gefolgt, sondern direkt von einer IO-Monade. Dieses entspricht der einfachen Sequenzialisierung. Wenn wir von einer I/O-Funktion kein Ergebnis erwarten, oder uns das Ergebnis egal ist, wie z.B. bei `putStr`, dann können wir diesen Operator benutzen.

Schließlich sehen wir in obigen Programmextrakt noch die Funktion `try`. Diese ermöglicht uns eine Fehlerabfrage. Das Ergebnis von (`try` *iofunction*) ist entweder: (`Left` *errormeldung*) oder (`Right` *ergebnis*).

5 Spezielle Syntax für Monaden

Zugegebener Maßen sieht obiges Programm noch etwas klobig aus. Deshalb wurde eine Syntax vorgeschlegen, die monadische Programme sehr anschaulich ausdrücken kann. Monadisches Programmieren besteht immer aus einer Sequenz von monasichen Ausdrücken, die durch `bind` verbunden sind. Die monadischen Ausdrücke können als eine Art imperative Befehle betrachtet werden. Es wird die *do*-Notation eingeführt:

```
Syntax:                          Bedeutung:
exp   -> do { stmt }
stmt  -> exp                     exp
stmt  -> exp ; stmt              exp >> do stmt
stmt  -> let decls ; stmt        let decls in do stmt
stmt  -> pat <- exp ; stmt       siehe unten
```

Im Prinzip ist `do` gefolgt von monadische Funktionen die `bind`-Sequenz von diesen Funktionen. Wir müssen noch klären, wie ein Ausdruck `pat <- exp` aufzulösen ist. Wir beschränken uns auf den Fall, wenn `pat` eine Variable ist:

```
     pat <- exp ; stmt
           → exp >>= \pat -> do
```

Zusätzlich, wie auch sonst in Haskell üblich, gibt es Lay-Out-Regeln für die *do*-Notation. Damit sieht unser obiges Programm wesentlich handlicher aus[4]:

```
main
  = do
     putStr "ZeilenEditor\n"
     newcommand []
```

[4] Programm konnte so noch nicht ausgetestet werden, da noch kein Haskellsystem alle dazu nötigen Leistungsmerkmale aufweist.

```
newcommand file
 = do
    eingabe <- readLine
    mainloop (eingabe,file)

readLine
  = do
     eof<- isEOF
     if eof then return [] else
        do
         c<-getChar
         if c `elem` ['\n'] then return [] else
            do
             cs<-readLine
             return (c:cs)
```

Literatur

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional languages. *The Computer Journal*, 32(2):108–121, 1989.

[Jon95] Mark P. Jones. Functional programming with overloading an higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 97–137. Springer, 1995.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, 1993.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 19th Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 24–52. Springer, 1995.