Merging Monads and Folds for Functional Programming

Konzepte von Programmiersprachen

Kurzzusammenfassung

Diese Ausarbeitung behandelt den gleichzeitigen Gebrauch von generalisierten Fold-Operatoren und Monaden, um Programme bei der funktionalen Programmierung zu strukturieren. Generalisierte Fold-Operatoren strukturieren Programme, indem die benutzten Werte abgebaut werden, und zwar sowohl bei der Bearbeitung vom Input als auch bei der Output - Überprüfung. Es wird gezeigt, wie generalisierte monadische Folds dazu beitragen, eine effiziente Graphenreduktion durchzuführen.

Valeri Kuznecov

Wintersemester 2003/2004

Inhaltsverzeichnis

Einleitung

Bei dem Programmieren stellt sich häufig die Frage, wann das Programm strukturiert werden soll, nachdem die Zerlegung von Werten erzeugt ist oder nachdem die Werte ausgerechnet sind? Es gibt verschiedene Meinungen darüber. In dieser Ausarbeitung soll gezeigt werden, dass die Syntax und die Semantik einer gezielten Programmierung geschickt kombiniert werden können.

1. Folding als Ersetzung der Konstruktoren durch Funktionen.

In den imperativen Programmiersprachen werden bevorzugt anstatt von nicht strukturierten **goto -** Schleifen die wiederholenden Konstruktoren wie **while-** und **for-** Schleifen benutzt. Auch bei der Programmierung in den funktionalen Sprachen ist es viel effizienter, die strukturierten rekursiven Operatoren anstatt von nicht strukturierten zu benutzen. Programme, die im Inhalt und Verhalten ähnlich sind, sollten auch in ähnlicher Form formuliert werden. Strukturierte Programme sind leichter zu verstehen und zugänglicher für Optimierung (auch für automatische) als ihre nicht optimierte Vorläufer.

Als nächstes sollen mehrere Funktionen betrachtet werden. Jede von diesen ist durch explizite Rekursion über Listen definiert. Wir werden sehen, dass alle diese Funktionen nach gleichem rekursivem Muster geschaffen sind. Sie können also so geschrieben werden, dass diese Ähnlichkeit leicht erkannt und ausgenutzt werden kann.

head

Funktion `head` gibt das erste Element von nicht leerer Liste zurück,

so `head [3, 2, 4] = 3.`

```
head :: [a] -> a
head (a:_) = a
```

filter

Funktion `filter` nimmt ein Prädikat `P` und eine Liste `as` und gibt nur diejenige Elemente von `as` aus, die das Prädikat `P` erfühlen. Z.B. `filter even [2, 3, 4] = [2, 4].`

```
filter             :: (a -> Bool) -> ([a] -> [a])
filter p []        = []
filter p (a:as)    = let as' = filter p as
                     in if p a then a:as' else as'
```

take

Funktion `take` nimmt eine ganze Zahl `n` und eine Liste `as` und liefert die Liste zurück, die aus ersten (höchstens) `n` Elementen der Liste `as` besteht. Z.B. `take 2 [4, 2, 6, 7] = [4, 2]`.

```
take               :: Int -> [a] -> [a]
take 0 _           = []
take _ []          = []
take (n+1) (a:as)  = a : take n as
```

foldl

Es sei eine binäre Funktion `f`, ein Akkumulator `b` und eine Liste `as` gegeben, z.B. `as = [4, 5, 1]`. Der Ausdruck `foldl f b as` wird ausgewertet als `((b `f` 4) `f` 5) `f` 1`. (Bemerkung: In Gofer kann man die Funktion in Einführungszeichen zwischen den Argumenten schreiben).

```
foldl              :: (b -> a -> b) -> b -> ([a] -> b)
foldl f b []       = b
foldl f b (a:as)   = foldl f (b `f` a) as
```

foldr'

Die Funktion `foldr` nimmt eine binäre Funktion `f`, ein Wert `a` und eine Liste `as` und liefert zurück den Wert, der so zustande kommt: in der Liste `as` werden der leere Konstruktor `[]` durch den Wert `b` und den Konstruktor `(:)` durch Operator `f` ersetzt. Z.B. `foldr f b [3, 2, 6] = 3 'f' (2 ,f' (6 ,f' b))`.

```
foldr              :: (b -> a -> b) -> b -> ([a] -> b)
foldr f b []       = b
foldr f b (a:as)   = a `f` (foldr f b as)
```

Jedes von diesen Beispielen ist induktiv über die Listen definiert und folgt dem rekursiven Muster der Funktion `foldr`. Der einzige Unterschied besteht darin, dass der Wert `b` für die leere Liste geliefert wird und die Funktion (`f`) das erste Element (`a`) der Liste mit dem rekursiven Aufruf von Funktion (`foldr`) auf den Rest der Liste verbindet.

Z.B. für head haben wir `head (a:as) = a ,f' (head as)`, wobei `f = \ a -> a`. Wir folgen daraus, dass wir die Funktion `head` durch `foldr` definieren können:

```
head = foldr (\ a _ -> a) (error „head []“) .
```

Für Funktion `filter p` haben wir

```
filter  p [] = [] für leere Liste
filter p (a : as) = a `f` (filter  p as )
```

für nicht leere Liste (a : as), wobei die Funktion f durch

```
f = \ a as' -> if p a then a : as'  else as'
```

definiert ist.

Daher haben wir:

```
filter p = foldr (\ a as' -> if p a then a:as' else as') [].
```

Zwei weitere Funktionen `take` und `foldl` in Termen von `foldr` auszudrucken, ist etwas umständlicher.

take'

Betrachte die Funktion `take n :: [a] -> a.` Im Gegensatz zur Funktion `filter` ist der erste Parameter von `take` nicht fixiert, d.h. `take (n + 1) (a : as)` ist in Termen von `take n as` definiert. Daher ist es unmöglich, `take n` in Termen von `foldr` direkt zu definieren. Der Trick dafür ist `swap` von beiden Argumenten der Funktion `take`:

```
take'           :: [a] -> Int -> [a]
take' []  = \ _  -> []
take' (a : as)= \n -> case n of 0->[]; n + 1 -> a : take' as n
```

Nun sehen wir, dass die Funktion `take` in Termen von `foldr` folgendermaßen ausdrückt werden kann:

```
take n as = foldr (\ a h -> \ n -> case n of o -> []; n + 1
                  -> a : h n) (\ _ -> [] ) as n
```

foldl'

Um die `foldl` in Termen von `foldr` auszudrücken, benutzen wir dieselbe Methode. Wir vertauschen zwei letzte Argumente von `foldl`:

```
foldl'               :: (b -> a -> b) -> [a] -> b -> b
foldl' f  []         = \ b -> b
foldl' f  (a : as)   = \ b foldl' f  as  (b `f` a)
```

Schließlich können `foldl` in Termen von `foldr` wie folgt ausdrückt werden:

```
foldl f b as = foldr (\a h -> \b ->h (b ,f' a)) (\b ->b) as b.
```

Die Empfehlung für Definition einer Funktion `h :: [a] -> b` durch `foldr (\ a b -> a `f` b) b` ist in folgende Richtung induktiv zu denken: „Angenommen, ich habe den Wert `b :: b` durch rekursive Anwendung der Funktion `h` auf den Rest von einer Liste `(a : as)` also `as` erhalten. Wie kombiniere ich diesen Wert `b` über eine Funktion `f :: a -> b -> b` mit `head` von Liste (also `a`), um den Ergebnis für `h (a : as)` zu erhalten?“ Es wird oft vorgeschlagen, eine geeignete Operation `f` durch die Typisierung der Betrachtungen zu suchen; für gegebene Werte `a :: a` und `b :: b` gibt es oft nur eine offensichtliche Wahl, um das Resultat `a `f ` b :: b` zu konstruieren. Man merke also, dass das Resultat der induktiven Anwendung von h eine Funktion sein könnte. In diesem Fall ist die gesuchte Operation `f` vom Typ `a -> (c -> d) -> (c -> d)` und konstruiert eine Funktion mit Parametern: Wert `a` und Funktion `(c -> d)`. Also ist Funktionale Programmierung die Programmierung mit Funktionen.

1.1 Optimierung des Programms durch Benutzen von foldr.

Der Funktional `foldr` erfüllt drei fundamentalen Regeln.

Identitätsregel

Die erste Regel erscheint ziemlich offensichtlich. Sie besagt, dass Folding von Listen mit Konstruktoren `[]` und `(:)` eine Identität auf Listen ist.

```
     foldr (:) []
= (Id)
     id.
```

Die Identitätsregel für Listen kann induktiv über Listen bewiesen werden.

Fusionsregel

Die zweite Regel für Funktion `foldr` ist als „Fusion low“ (Fusionregel/Verschmelzungsregel) bekannt. Diese Regel bestimmt die Bedingungen, unter welchen die per Folding produzierenden Zwischenwerte eliminiert sein können.

```
     h.foldr f b = foldr g (h b)
<= (Fusion)
     h (a `f` b)  = a `g` (h b)
```

Fusion ist „free theorem“. Das kann man per Induktion über Listen beweisen. Für unendliche Liste wird angefordert, dass `h` strickt ist. Hier werden wir überwiegend die endlichen Listen betrachten.

Als eine einfache Anwendung von Fusion kann gezeigt werden, dass Multiplikation stärker als Summieren der Elemente von Liste bindet.

```
(n*) . foldr (+) 0 = foldr ((+) . (n*)) 0          (1)
```

Dieses Beispiel ist etwas übertrieben, da der Ausdruck auf der linken Seite weniger Multiplikationen enthält als der Ausdruck von rechts; manchmal ist es sinnvoll, die Zwischenwerte anzufügen anstatt diese zu entfernen .

Aus (1) folgt:

```
     (n*) . foldr (+) 0 = foldr ((+) . (n*)) 0
<= (Fusion)
     n * (a + m) = n * a   +   n * m   ∧   n * 0 = 0
≡ Arithmetik
     TRUE
```

Die Bedingung `n * 0 = 0` ist durch Übereinstimmung von `folds` im Beispiel mit den `folds` in der Fusion entstanden.

Acid Rain Theorem

Die dritte Regel ist bekannt als „Acid Rain Theorem“ oder foldr/build – Regel. Diese Regel gibt die Bedingungen an, unter welchen die Zwischenwerte eliminiert werden können, die durch Folding konsumiert wurden. Also, wir erhalten „deforestation for free“ („kostenlose Abholzung“), was den Namen vom Theorem erklärt.

```
foldr f b . d (:) [] = g f b
<== (Acid Rain)
g :: (A -> b -> b) -> b -> (C -> b)
```

für fixierte Typen `A` und `C`.

Acid Rain folgt aus dem „free theorem“ für die listenerzeugende Funktion `g`. Dies führt zu der Annahme, dass man in der ersten Erstellung einer Zwischenliste, die `g (:) []` benutzt, `(:)` durch die Funktion `f` und `[]` durch den Wert `b` ersetzen kann. Dadurch erreicht man dasselbe Resultat, als wenn man direkt die Funktion `f` und Wert `b` benutzt.

Als eine einfache Anwendung von Acid rain zeigen wir, dass die Abbildung der Funktion über Liste die Länge dieser Liste nicht ändert.

```
length . map h = length        (2)
```

Die Funktionen `length` und `map h` können beide durch `foldr` wie folgt definiert werden:

```
map :: (a->b)->([a]->[b])
map h =foldr ((:) . h) []

length :: [a] -> Int
length = foldr (\ _ n -> n +1) 0
```

Um die Gleichung (2) zu beweisen, merken wir uns, dass `map h = g h (:) []`, wo `g h f b = foldr (f.h) b` und `g h :: (a->b->b)->b->([c]->b)`, wobei `h :: c->a` (*). Der aktuelle Beweis ist jetzt eine einfache Rechnung:

`length .map h`
= Definition von `length`, Bemerkung (*)
`foldr (\ _ n -> n+1) 0 . g h (:) []`
= (*), Acid Rain
g h (\ _ n -> n+1) 0
= Definition von `g` und `length`
`length`

Also, die Gleichung (2) kann per Fusionsregel bewiesen werden.

1.2 Konstruktive Algorithmik ist Rechnung mit Programmen

Fusion und Acid Rain machen durch ihre Spezifikation bei Programmrechnung einen Sinn. Die Ingenieure lösen dieses Problem oft durch Anwendung von Integral- und Differenzialrechnung. Ähnlich sind Computerprogramme in „Construktive Algoritmics“ mittels solcher Programmanalyse konstruiert. Die zentrale Idee ist es, mit Programmen zu rechnen. Spezifikationen und Programme werden formal als ein Objekt in denselben theoretischen Gerüsten und als ein Subjekt in denselben Rechnungsregeln betrachtet. Das Problem der Programmkorrektheit reduziert sich auf korrekte Anwendung der einfachen Rechnungsregeln:

Spezifikation
= law
…
= law
Implementierung

Die meisten der üblichen Ansätzen, um ein richtiges Programm zu schreiben, basieren auf einer Methode, über Programme logisch zu denken. Deshalb sind zwei Sprachen notwendig: eine Programmiersprache und eine Sprache, mit der man über Programme logisch denken kann. Programmkalkulation benötigt nur eine Sprache. Um die Kalkulationsregeln anzuwenden, sollten diese leicht zu merken und zu manipulieren sein. Das ist genau der Grund, warum die Oberflächenaspekte im Bereich der Konstruktiven Algorithmik eine wichtige Rolle spielen. Eine große Menge von Regeln ist unabdingbar für eine effektive Rechnung mit Programmen.

2. Verallgemeinerungen von Folding auf andere Typen.

Folding existiert nicht nur für Listen. Andere Datentypen können auch durch Folding realisiert werden, wobei ihre funktionellen Konstruktionen durch die Funktionen von geeigneten Typen ersetzt werden.

2.1 Snoc-Listen

Unter dem Namen foldr versteht man „Faltung von rechts“, also werden die Elemente von rechts nach links bearbeitet. Der Name foldl meint „Faltung von links“, indem man ein Bearbeitungsprozess von links nach rechts aufruft. Das ist oft zweckmäßiger, weil beim Ausrechnen von i-tem Element die ersten i-1 Elemente schon verfügbar sind. Wie gezeigt wurde, ist die Funktion foldl kein natürlicher Faltungsoperator über Listen, eher entspricht sie einer Funktion, die so genannte Snoc-Liste faltet. Snoc-Listen sind von links nach rechts aufgebaut.

```
data Snoc a = II | (Snoc a) :%: a
```
(Vergleich: `data Cons a = [] | a : (Cons a)`)

Um die Snoc-Liste zu falten, ersetzen wir rekursiv den konstanten Konstruktor `II : Snoc a` durch eine Konstante `nil:: b` und Funktion `(:%:) :: Snoc a -> a -> Snoc a` durch Funktion `snoc :: b -> a -> b`.

```
foldS :: b ->(b->a->b) -> (Snoc a -> b)
foldS  nil snoc = fS
     where  fS II = nil
            fS (as :%: a) =  snoc (fS as) a
```

Genau wie `fold`-Operator auf `cons`-Listen ist das der Fall, dass das Folding die Identität auf `snoc`-Listen ist, das die Funktionale Konstruktion benutzt.

```
foldS  II  (:%:) = id
```

Also, die Funktion `foldS` erfühlt eine Fusionsregel und ein Acid Rain Theorem. Die Fusionsregel für `snoc`-Liste ist gegeben durch:

```
          h . foldS b f = folds (h b) g
```
<= (Fusion)
```
          h (b ,f' a) = (h b) ,g' a
```
wobei das Acid Rain Theorem ist gegeben durch:
```
     (foldS b f) . (g  II (:%:)) = g b f
```
<= Acid Rain
`g:: b -> (b -> A -> b) -> (C -> b)` für fixierte Typen `A` und `C`

2.2 Peano Zahlen

Es gibt zahlreiche Wege, die Natürlichen Zahlen als einen induktiv definierten Typ zu betrachten, auch wenn sie nicht expliziert definiert sind. Eine mögliche Sicht ist Peano – Sicht. Die Zahl ist eins von beiden: `zero`, d.h. 0 oder die Funktion `successor (n+1)` von Zahl `n`. Die Faltung einer Zahl `n` durch Konstante `zero` und `succ`-Funktion wird zur n-maligen Anwendung der Funktion `succ` auf Konstante `zero` reduziert, also `foldN zero succ n = succ(…(succ zero))`.

```
     foldN :: a-> (a->a) -> (Int -> a)
     foldN  zero succ = fN
     where  fN 0 = zero
          fN (n+1) = succ (fN n)
```

Die Fusionsregel für Faltung der realen Zahlen wird definiert als:

```
     f . (foldN a g)  = foldN (f a) h
```
<= Fusion
```
     f (g a) = h (f a)
```

Das Acid Rain Theorem ist definiert als:

```
     (foldN a f ) . (g  0  (+1)) = g a f
```
<= Acid Rain
`g :: a→  (a→ a) →  (B →  a)` für ein einfache fixierte Typ `B`

Zusammen mit dem Identitätsaxiom für Zahlen foldN 0 (+1) = id impliziert Fusion die Induktion über Zahlen. Sei p :: Int → Bool ein Prädikat über natürliche Zahlen, dann

```
      \ n → (n, p n)
```
= (Id)
```
      (\ n → (n, p n)) . foldN (0, (+1))
```
= (Fusion), Annahme/Voraussetzung: p (n+1) = p n || p (n+1)
```
      foldN ((0, p 0), \ (n,b) →  (n+1, b || p (n+1)))
```
= (Fusion), Annahme: p 0 = True
```
      (\ n →  (n, True)) . foldN (0, (+1))
```
= (Id)
```
      \ n → (n, True)
```

Daraus können wir schließen, dass das Prädikat `p` für alle natürlichen Zahlen `true` ist, falls `p` für Zahl 0 `true` ist und `p (n+1)` aus `p n` folgt.

```
      p = \ n →  True
```
<= (Induktion)
```
      (p (n+1) = p n || p (n+1) ) ^ (p 0 = True)
```

Viele bekannte arithmetische Funktionen können durch Folding definiert werden. Z.B. Addition und Multiplikation:

```
      (<+>), (<*>) :: Int -> Int -> Int
      n <+> m = foldN (m, (+1)) n
      n <*> m = foldN (0, (<+> m)) n
```

Die Addition n zu m beginnt mit m und erhöht m um 1 genau n mal. Die Multiplikation m mit n startet mit 0 und erhöht sie um m genau n mal.
Der Beweis, dass die Multiplikation über die Addition distributiert, zerfällt in zwei Schritte, der eine folgt aus Fusion, der andere aus Acid Rain.

```
      (n <+> m) <*>
```
= (Acid Rain)
```
      foldN (m <*> k, (<+> k )) n
```
= (Fusion)
```
      (n <*> m) <+> (m <*> k)
```

Der zweite Schritt folgt unmittelbar aus Fusion durch die Annahme, dass Addition <+> kommutativ ist. Der erste Schritt ist interessanter, da er zeigt, dass die Prämisse von Acid Rain Theorem weniger intuitiv sein kann.

```
      (n <+> m)  <* > k
```
= reorganisieren, Definition von <+>
```
      (<+> k)  (foldN (m, (+1)) n )
```
= definiere: `g = \ (a,f) -> foldN (foldn (a,f) m, f ) n`
```
      (<*>  k) (g (0, (+1)))
```
= Definition von `(<+> k)`

```
      foldN  (0, (<+> k)) (g (0,(+1)))
= Acid Rain
      g (0, (<+> k))
= Definition von g
       foldN (foldN (0, (<+> k)) m, (<+> k)) n
= Definition von (<*>)
      foldN (m <*> k, (<+> k)) n
```

2.3 Binary trees

Alle Datentypbeispiele, die wir gesehen haben, sind linear strukturiert. Unser nächstes Beispiel für einen Datentyp, der verzweigte Struktur hat, ist ein Binärbaum. Der Binärbaum besteht entweder aus einem Blatt, das den Wert vom Typ a hat, oder aus einer Wurzel, die einen linken und einen rechten Teilbaum hat.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Folding der Bäume folgt dem gleichen Muster wie Folding von Listen und Zahlen, indem die Konstruktoren durch die Funktionen ersetzt werden.

```
foldB :: (a -> b, b -> b -> b) -> (Tree a -> b)
foldB (leaf, node) = fB
     where
          fB(Leaf a) = leaf a
          fB (Node as as') = node (fB as) (fB as')
```

Genau wie es mit `foldr` und `foldN` gemacht wurde, können wir eine Menge von Funktionen in Termen von Baumfaltung mit `foldB` definieren statt explizit Induktion zu benutzen.

2.4 Rose Trees (Rhododendrons)

Es ist noch interessanter, wenn ein Datentyp durch andere Typen definiert ist, wie z. B. Datentyp Rose, ein mehrfach verzweigter Baum. Die Wurzel von Rose hat die Liste von Teilbäumen:

```
data Rose a = Fork a [Rose a]
```

Der Datentyp Rose benutzt in seiner Definition den Datentyp Liste []. Die Hauptidee ist, dass die Funktion map in `foldR` rekursiv auf die Elemente von der Liste der Teilbäume angesetzt wird.

```
foldR :: (a -> [b] -> b) -> (Rose a -> b)
foldR (fork) = fR
     where fR (Fork a ts) = fork a (map fR ts)
```

Als ein Beispiel für fold über Rose trees ist Funktion `postOrder :: Rose a -> [a]`, die die Werte von tree in Postorder aufzählt:

```
postOrder :: Rose a -> [a]
postOrder = foldR (\ a ass -> (concat ass) ++[a])
```

2.5 Typisierte λ- Ausdrücke.

In diesem Teil wird die Faltung für etwas komplizierten Datentyp definiert nämlich für λ-Ausdrücke.
Ein einfacher Typ ist Variabeltyp oder Funktionstyp.

```
data Type = TVar String | Type :->: Typ
```

Typisierter λ- Ausdruck ist Variable, Applikation oder eine typisierte Anwendung, die den Typ mit gebundenen Variablen assoziiert.

```
data Expr = Var String | Expr :@: Expr | (String, Type) ::: Expr
```

Die Faltung für Typen und Expr wird wie folgt definiert:

```
foldT :: (String-> a, a->a->a)->(Type->a)
foldT (tvar, arrow) = fT
     where  fT (Tvar s) = tvar s
          fT (s :->: t) = arrow (fT s) (fT t)

foldE ::(String->b, b->b->b,(String, Type)->b->b)->(Expr-> b)
foldE (var, apply, lambda) = fE
     where     fE (Var x) = var x
          fE (f :@: a) = apply (fE f) (fE a)
          fE ((x,t) ::: b) = lambda (x,t) (fE b)
```

2.6 Wechselseitige rekursive Datentypen.

Faltung kann auch für wechselseitige rekursive Datentypen definiert werden. Z. B. für ein Paar: Zig und Zag. Diese Typen können für das Implementieren von Listen benutzt werden, in denen der Typ von Elementen alternierend ist. Also:

```
data Zig a b = Nil | Cins a (Zag b a)
data Zag a b = Nal | Cans a (Zig b a)
```

Die Faltung von Zig und Zag ist auch wechselseitig rekursiv. Folds für Zig und Zag nehmen zwei Paaren als Argumente: das eine entspricht dem Zig –Typ, das andere entspricht dem Zag - Typ.

```
foldZig :: (c, a->d->c) -> (d, b->c->d) -> (Zig a b -> c)
foldZig (nil, cins) (nal, cans) = fZig
     where  fZig Nil = nil
            fZig (Cins a z ) = cins a (fZag z)

            fZag = foldZag (nil, cins) (nal, cans)


foldZag :: (c, a -> d -> c) -> (d, b -> c -> d) -> (Zag b a
-> d)
foldZag (nil, cins) (nal, cans) = fZag
     where  fZag  Nal = nal
            fZag (Cans a z) = cans a (fZig z)

            fZig = foldZig (nil, cins) (nal, cans)
```

3. Monaden trennen Werte und Rechnung.

So viel zum Programmieren mit folds. Nun möchten wir uns dem Strukturieren von Programmen mit einer bestimmten Methode zuwenden, und zwar nach der Schätzung von ihren resultierenden Werten.

3.1 Anwendungen.

Angenommen, wir sollen eine Funktion schreiben, die den Gesamtwiderstand einer Liste von parallel geschalteten Widerständen berechnet.

```
resistance :: [Float] → Float
resistance = (1 /) . foldr (\ r r' → 1/ r+r') 0
```

Wenn Funktion `resistance` auf die leere Liste angewendet wird oder wenn ein Widerstand in der Liste Null ist, dann wird die Ausführung des Programms, in dem diese Funktion aufgerufen wird, abgebrochen, und ein Fehler wird vom Gofer Interpreter ausgegeben: `Program error: {primDivInt 13 0}`. Ein solches Verhalten ist sicherlich beim Programmnutzen nicht erwünscht, daher muss dagegen etwas getan werden, damit die Nutzung der Funktionellen Programmierung sich weiter verbreitet.

Eine bessere Methode ist es, diese Funktion signalisieren zu lassen, falls eine Division durch Null vorkommt.

Um Exceptions in die Berechnung aufzunehmen, führen wir bei der Auswertung von `Typ a` einen neuen Typ `Maybe` ein.

```
data Maybe a =  No | Yes a
```

Dann ändert sich der Typ der `resistance` – Funktion zum Typ

```
resistance : : [Float] → Maybe Float.
```

Dies führt dazu, dass die Berechnung entweder abgebrochen und eine Exception `No` zurückgegeben wird, oder die Berechnung wird mit der Rückgabe eines bestimmten Wertes (z.B. `Yes r`) abgeschlossen. Nach der Änderung des Funktionstyps, muss auch die Funktionsdefinition geändert werden. Bei jeder rekursiven Anwendung der `resistance` – Funktion wird das Ergebnis durch Fallunterscheidung geprüft. Außerdem wird eine Exception hochgezogen. (z. B. Division durch Null).

```
resistance :: [Float] → Maybe Float
resistance
     = (\ mr → case mr of
               No    → No
               Yes r → if r == 0 then No else Yes (1/r)
      )
     .foldr (\ r mr' →case mr' of
                         No →No
                         Yes r' →if r == 0 then No
                                   else Yes (1/r +r')
                )
               (Yes 0)
```

Es gibt vier Fälle:

1. Hochziehen einer Exception mit Ausgabe von `No`;
2. Ausgabe eines bestimmten Wertes, der in `Typ Maybe` mittels der Konstruktorfunktion Yes injektiert ist;
3. Weiterleiten von zwei Alternativen von `Typ Maybe`;
4. Test des zweiten Operanden.

Diese wiederholenden Muster werden entsprechend `zeroM`, `resultM`, `bindM` und `divM` genannt.

```
zeroM :: Maybe a
zeroM = No

resultM :: a →Maybe a
resultM a = Yes a

bindM :: Maybe a → (a → Maybe a) → Maybe b
bindM ma f = case of No → No; Yes a → f a

divM :: Float → Float → Maybe Float
divM r r' = if r' == 0 then zeroM else resultM(r/r')
```

Durch den Gebrauch dieser Funktionen wird die Exceptionsbehandlung strukturiert: ungeschickte Fallunterscheidung ist endgültig ausgedrückt und wird im Programmtext nicht mehr wiederholt.

```
resistance :: [Float] → Maybe Float
resistance rs
= bindM (foldr (\r mr'→ bindM mr'
                                (\r'→ bindM (div 1 r)
                                        (\r'' → resultM (r''+r'))
                                )
                )
                (resultM 0)
                rs
          )
          (\ r → divM 1 r)
```

3.2 Blätterersatz in Baum.

Eine Assoziationsliste ist eine Liste von Schlüssel/Wert-Paaren. Der Wert wird durch die Ausführung der Funktion `lookup` dem Schlüssel zugewiesen. Im Fall, dass ein Schlüssel mit keinem Wert assoziiert ist, gibt `lookup` eine Exception aus.

```
data Assoc a b = a := b

lookup    :: Eq a ⇒ a → [Assoc a b ] → Maybe b
lookup a table = case [b | a' := b ← table, a' == a] of
                      [] → zeroM
                      (a : _)→ resultM a
```

Als nächstes wird das folgende Problem betrachtet. Gegeben seien ein binärer Baum und eine Assoziationsliste. Es sind die Werte im Baum durch ihre Assoziationswerte zu ersetzen. Da der Baum rekursiv traversiert werden muss, um die Einträge in den Blättern zu ersetzen, wäre es sinnvoll eine Ersetzungsfunktion zu definieren, die das Folding foldB benutzt. Da die Ersetzung scheitern kann, weil einige Blätter keine Assoziationswerte haben, wäre es sinnvoll `resultM` und `bindM` zu benutzen.

```
subst :: Tree a → ([Assoc a b ] → Maybe (Tree b))
subst
= foldB (\a → bindM (\table → lookup a table)
                    (resultM (Leaf b)),
          \ mbs mbs' → bindM (\ table → (mbs table))
                       (\bs → bindM (mbs' table)
                              (\bs' →
                              resultM (Node bs bs'))
                               )
          )
```

Hier haben wir ein anderes wiederholendes Muster, das wir im Folgenden abstrahieren können: die Tabelle überliefert explizit die rekursiven Aufrufe von `subst` – Funktion. Das nächste Ziel ist es, die `subst` – Funktion so umzuschreiben, dass das Argument `table` eliminiert wird. Für diesen Zweck werden zwei neue Operationen `resultR` und `bindR` definiert, die die vorhandenen Operationen `resultM` und `bindM` allein durch das Überziehen eines Arguments erweitern. Anschließend wird gefordert, dass die Operation `zeroM` dieses neue Argument akzeptiert. Um Lesbarkeit zu erhöhen, wird die Abkürzung `ReaderM r a` statt `r → Maybe a` benutzt.

```
type ReaderM r a = r → Maybe a

resultR :: a → ReaderM r a
resultR a = \ _ → resultM a

bindR :: ReaderM r a → (a → ReaderM r b) → ReaderM r b
bindR ma f = bindM (\ r → ma r) (\ a → f a r)

zeroR :: ResderM r a
zeroR = \ _ → zeroM
```

Durch Überschreiben von Funktion `subst` mit neudefinierten Operationen, bekommt man die bessere Prägnanz für `subst`.

```
subst :: Tree a → ReaderM [Assoc a b] (Tree b)
subst
= foldB (bindR (\ a → (lookup a)
                (\ b → resultR (Leaf b)),
         bindR (\ mbs mbs' → mbs)
                (bindR (\ bs → mbs')
                       (\ bs' → resultR (Node bs bs'))
                )
          )
```

3.3 Typchecks

Ein typischer λ-Ausdruck ist vom korrekten Typ, wenn die Typen von gebundenen Variablen im Ausdruck mit den Typen übereinstimmen, die im Prozess erforderlich sind, in dem die gebundenen Variablen benutzt werden. Funktion `flip (|-) :: Expr → ReaderM [Assoc Var Type] Type` versucht den Typ von einem Ausdruck aufzubauen, der als Basis gegeben ist, die Variablen mit ihren deklarierten Typen assoziiert.
Bei Überprüfung einer Abstraktion wird die Basis zu einem Datensatz erweitert, dessen Typ mit dem Typ von gebundenen Variablen der Abstraktion assoziiert wird. Für diesen Zweck werden die Funktionen `fetchR` und `restoreR` eingeführt, sodass Basis als Resultat zurückgegeben und die Berechnung mit einer neuen Basis fortgesetzt wird.

```
fetchR :: ReaderM r r
fetchR = \ r → resultM r
```

```
restoreR :: r → ReaderM r a → ReaderM r a
restoreR r ma = \ _ → ma r
```

Bei der Überprüfung des Typen einer Variable wird deren Typ auf der Basis abgelesen. Um Typ einer Abstraktion `(x, s) ::: b` zu überprüfen, wird Typ von `b` überprüft mit der Voraussetzung, dass `x` vom Typ `s` ist. Wenn der resultierende Typ `t` ist, dann ist die ganze Abstraktion von Typ `s :→: t`. Um Typ einer Applikation `f :@: a` zu überprüfen, soll vergewissert werden, dass `f` eine Funktion von Typ `s :→: t` ist und dass der Typ `s'` von `a` das gleiche ist wie der Argumenttyp `s`, der bei `f` erwartet wird.

```
env |-expr
= foldE (\ x → lookup x,
         bindR (\ f a → f)
               (\ s → case  s of
                      (bindR ((s :→: t) → a)
                             (\ s' →   if s == s'
                                       then resultR t
                                       else zeroR)
                       ( _ → zeroR),
         bindR (\ (x,s) b → fetchR)
               (bindR (\ env → restoreR (x:= s : env) b)
                      (\ t → resultR (s : →: t)),
         bindR (\ (x,s) b → fetchR)
               (bindR (\ env → restore))              )
```

3.4 Zerteilte Bäume, markierte Knoten

Ein gerichteter Graph mit Knoten von Typ `a` kann durch einen Wert von Typ `Rose a` repräsentiert werden. Zum Beispiel, ein kleiner Graph mit neun Knoten ist gegeben wie folgt:

```
root = Fork „root" [a,b,c,d,e,f,g,h,i,j]
     where
     a = Fork „a" [g,j];   b = Fork „b" [a,i]
     c = Fork „c" [e,h];   d = Fork „d" []
     e = Fork „e" [d,h,j]; f = Fork „f" [i]
     g = Fork „g" [f,b];   h = Fork „h" []
     i = Fork „i" [];      j = Fork „j" []
```

Die Aufgabe der Funktion `dff` („depth first forest“) ist es, alle Kanten zu den Knoten zu eliminieren, die bei der ersten depth-first-Suche vom Graphen abgecheckt wurden, so das der originale aufgespannte Graph bleibt. Für den oberen Graphen liefert die Funktion `dff` den Graph:

```
root = Fork „root“ [a,c]
     where


     a = Fork „a“ [g,j];    b = Fork „b“ []
     c = Fork „c“ [e];      d = Fork „d“ []
     e = Fork „e“ [d,h];    f = Fork „f“ [i]
     g = Fork „g“ [f,b];    h = Fork „h“ []
     i = Fork „i“ [];       j = Fork „j“ []
```

Funktion `dff` ist durch eine Hilfsfunktion `chop` definiert. Während der Traversierung des Graphen wird eine von tatsächlich besuchten Knoten erstellt. Um abzusichern, dass die aktualisierte Liste von besuchten Knoten in die Berechnung einbezogen wurde, liefert Funktion `chop` die vergrößerte Liste als Teil der Antwort zurück. Da Teilbäume in `dff` nicht vorkommen können, liefert Funktion `chop` einen Baum als Sonderfall in der anderen Hälfte ihrer Antwort. So hat Funktion `chop` Typ Rose a → ReaderM [a] (Maybe (Rose a), [a]). Funktion `dff :: Rose a → Rose a` ruft `chop` mit einer leeren Liste von besuchten Knoten auf und eliminiert die endgültige Liste von allen erreichten Knoten, die von Funktion `chop` geliefert wurde.

```
dff g = g’ where Yes (Yes g’,_) = chop g []
```

Es wird nicht gleich versucht, Funktion `chop` mit Hilfe von `ReaderM` - Operation zu schreiben. Statt dessen wird Typ `ReaderM s (a,s)` als `StateM s a` abgekürzt und die Operationen `resultR`, `bindR` und `zeroR` werden aufgehoben und bei der Arbeit mit Typ `StateM s a` eingesetzt.

Um die Liste von besuchten Knoten zu manipulieren, werden die Operationen `peekS` und `pokeS` vorgeschlagen.

```
resultS :: a → StateM s a
resultS a = bind fetchR (\s → resultR (a,s))

bindS :: StateM s a → (a → State s b) → StateM s b
bindS ma f = bindR ma (\ (a,s) → restoreR s (f a))
```

```
zeroS :: StateM s a
zeroS = zeroR

peekS :: (s → a) → StateM s a
peekS f = bindR fetchR (\ s → resultR (f s, s))

pokeS :: (s → s) → StateM s ()
pokeS f = bindR fetchR (\ s → resultR ((), f s))
```

Um einen Knoten zu kürzen, wird überprüft, ob ein Knoten zuvor besucht worden ist. Wenn ja, wird der Knoten gekürzt. Wenn nein, wird der Knoten als besucht markiert, der Teilbaum vom Knoten wird rekursiv geteilt und es wird der Knoten zurückgeliefert, der nur nicht gekürzte Teilbäume als Kinder enthält.

```
chop :: Rose a → StateM [a] (Rose a)
chop
= foldR (bindS (\a mts → peekS (elem a))
               (\v → if v then resultS No
                          else
                            bindS (pokeS (a:))
                                  (bindS (accumulateS mts)
                                    (\ts →
                                    resultS (Yes (Fork a
                                                  [t|Yes t ← ts]))
                                  )
               )
        )
```

Funktion `accumulateS :: [StateM s a] → StateM s [a]` sammelt die Resultaten von Listenkürzung von Teilbäumen aus left – to – right in eine Resultatliste.

```
accumulateS :: [StateM s a] → StateM s [a]
accumulateS = foldr (bindS (\mt mts → mt)
                           (bindS (\t → mts)
                                  (\ts → result (t:ts))
                           )
                    )
```

3.5 Monaden generalisiert bind und result

Sei eine Funktion `f` von Typ `a → b` und ein Konstruktor `m` gegeben (wie `Maybe`), dann kann `f` zu einer monadischen Funktion `f :: a → m b` generalisiert werden, die auf Elementen von Typ `a` definiert ist und das Resultat von Typ `b` mit einem möglichen Nebeneffekt vom Konstruktor `m` zurückliefert.

Um monadische Funktionen effektiv zu nutzen, werden eine Operation `bind :: m a → (a → b) → m b` und ein Funktion `result :: a → m a` gebraucht. Operation `bind` bindet eine Funktion `f` von Typ `a → m` an ein Argument `ma` von Typ `m a`. Funktion `result` liefert ein Ergebnis von Typ `m a`. Dann ist die Klasse Monad m definiert als:

```
class Monad m where
     result :: a → m a
     bind   :: m a → (a → m b) → m b
```

Für `bind` und `result` sollen right-unit und left-unit Regeln erfüllt werden:

```
bind k result = k
bind result a k = k a
```

Dabei muss `bind` assoziativ sein. Für jede Instanz von der Klasse `Monad` müssen diese Regeln verifiziert sein.

3.6 Ausnahmen

Wie weiter unten gezeigt ist, ist Datentyp `Maybe` eine Instanz von Klasse `Monad`:

```
instance Manad Maybe where
     result a = Yes a
     bind ma f = case ma of No → No; Yes a → f a
```

Außerdem kann `bindM` durch Folding über den Type `Maybe` definieret werden. Tatsächlich kann die Operation `bind` für viele Monaden `m` als ein über `m a` generalisiertes Fold definiert werden.

Die Ausnahmen sind an Beispielen von Monaden mit `zero` zu sehen. Für `zero :: Monad m ⇒ m a` gilt, dass `zero ´bind´ f = zero` ist.

```
class Monad m => MonadO m where
     zero :: m a

instance MonadO Maybe where
     zero = No
```

Ausnahmen sind Beispiele von Monaden mit einem Plus (++). Der zweistellige Operator `(++) :: MonadO m ⇒ m a → m a → m a` so dass `zero ++ ma = ma` und `ma ++ zero = ma`.

```
class MonadO m ⇒ MonadPlus m where
     (++) :: m a → m a → ma

instance MonadPlus Maybe where
     ma ++ ma´ = case ma of No →  ma´ ; _ → ma
```

3.7 Readers

Die Monade von *state readers* bringt ein Extraargument auf die Spitze einer Basismonade. Wir wiederholen den Typ von Reader.

```
type Reader m r a = r →  m a

instance Monad m ⇒ Monad (Reader m r) where
     result a = \ _ →  result a
     bind ma f = bind (\ r → ma r) (\ a → f a r)

instance MonadO m ⇒ MonadO (Reader r m) where
     zero = \ _ → zero

instance MonadPlus m ⇒ MonadPlus (Reader m r) where
     ma ++ ma´ = \ r → ma r ++ ma´ r
```

die Reader – Monade bietet zwei Operationen an, um das Argument abzurufen und zu ersetzen.

```
fetch :: Monad m ⇒ Reader m r  r
fetch = \ r →  result r

restore :: Monad m ⇒ r → Reader m r a → Reader m r a
restore r = \ ma → \ _ → ma r
```

3.8 Zustandstransformationen

Die Monade von Zustandstransformationen kann als eine Extraebene der Reader-Monade betrachtet werden.

```
type State m s a = Reader m s (a, s)

instance Monad (Reader m s) ⇒ Monad (State m s) where
     result a = bind fetch (\ s → result(a,s))
     bind ma f = bind ma (\ (a,s) → restore s (f a))

instanve MonadO (Reader m s) ⇒ MonadO(State m s) where
     Zero = zero

instance MonadPlus (Reader m s) ⇒ MonadPlus (State m s)
where
     (++) = (++)
```

Zwei addierende Operationen auf die Zustandsmonade sind peek und poke. Diese Operationen werden benutzt, um den Zustand zu inspizieren und zu modifizieren.

```
peek :: Monad (Reader m s) ⇒ (s → a) → State m s a
peek f = bind fetch (\ s → result(f s,s))

poke :: Monad (Reader m s) ⇒ (s → s) → State m s ()
poke f = bind fetch (\ s →  result ((),f s))
```

3.9 do-Notation

Die Ausdrücke können mithilfe von do-Notation wie folgt definiert werden:

expr ::= `do{`*qualifer; ...; qualifer; expr*`}`
qualifer ::= *pat* ← *expr* | *expr* | `if` *expr*

Für *e* (*expr*), *p* (*pattern*) und *gse* (*qualifier* - Sequenz) gilt:

```
do{e}              =    e
do{p ← e; qse }    =    bind e f where f p =  do{qse}; f _ = zero
do{e; qse}         =    bind e (\ _ →  do {qse})
do{if e; qse}      =    if e then do{qse} else zero
```

Mithilfe von do-Notation bekommen wir als Endergebnis:

```
env | - expr
= folfE (\x    →  lookup x
     , \f a    →  do { (s :→ : t) ← f
               ; s' ← a
               ; if s == s'
               result t
               }
     , \(x,s) b → do{ env ← fetch
               ; t  ← restore (x : = s : env) b)
               ; result (s : → : t)
               }
     ) expr env
```

4. Monadic folds

Im folgenden Abschnitt soll behandelt werden, wie Monaden und Folds kombiniert genutzt werden. Es werden zwei Beispielsfunktionen betrachtet.

Die Funktion mapr bildet eine monadische Funktion über Listen ab und startet von rechts. Im Vergleich zur normalen map ist bei Monaden-Rechnung die Reihenfolge wichtig, in der die Anwendung ausgeführt wird.

```
mapr :: Monad m ⇒ (a →  m b= →  ([a] → m [b])
mapr f
= foldr (\a mbs →  do{bs ← mbs; b ← f a; result (b:bs)})
        (result [])
```

Wenn die Funktion resistance (s. Abschnitt 4.1) mithilfe von do-Notation umgeschrieben wird, sieht sie wie folgt aus:

```
resistance :: [Float] → Maybe Float
resistance rs
= do{s ← foldr (\ r mr' →  do{ r' ← mr';
                        r'' ← 1 `divM` r;
                        result (r'' + r')
                        }
           )
          (result 0) rs;
     1 `divM` s
     }
```

Die Funktionen `mapr` und `resistance` benutzen beide für Folding die Konstruktorfunktion (:), die wie folgt aussieht:

```
\ a mb → do{b ← mb; mcons a b}
```

Weiter wird das Folding für Listen mit Monaden definiert:

```
mfoldr :: Monad m ⇒ (a → b → m b) → m b → ([a] → m b)
mfoldr mcons mnil
= foldr (\ a mb → do{b ← mb; mcons a b}) mnil
```

Es kann gezeigt werden, dass diese Funktion den Regeln folgt, die im Abschnitt 2.1 dargestellt sind.

1. Die Identitätsregel für monadische Folds besagt, dass monadische Folding einer Liste mit monadischen Konstruktoren mit Einführen einer Liste in die Monade direkt äquivalent ist:

```
   mfoldr (\a as → result (a:as)) (result []) as
= (Id)
   result as
```

2. Die monadische Fusionsregel besagt, dass der monadische Aufbau einer Funktion, die monadisch über die Funktion verteilt, die Konstruktorfunktion `cons` durch ein monafisches Fold ersetzt, wieder ein monadisches Fold ist:

```
   do{b ← mfoldr f mb as; h b} = mfoldr g (do{b ← mb; h b}) as
<= (Fusion)
   do{c ← mc; b ← f a c; h b} = do{c ← mc; b ← h c; g a b}
```

3. Das monadische Acid-Rain-Theorem besagt, dass es notwendig ist, erst eine Liste mit einer Monade zu bilden und dann diese zu falten, vorausgesetzt die Funktion, die die Liste bildet, polymorph genug ist:

```
   do{as ← g (\ a as → result (a:as)) (result []) c
        ; mfoldr f n as} = g f n c
   = (Acid Rain)
   g :: Monad m ⇒ (A → b → m b) → m b → C → m b
```

   für fixierten Typen A und C.

Ein Beispiel für eine Funktion, die mithilfe eines monadischen Folds nicht geschrieben werden kann, ist Funktion `mapl`. Die Funktion `mapl` bildet eine monadische Funktion über Liste, indem sie vom Links startet.

```
mapl :: Monad m ⇒ (a → m b) → ([a] → m [b])
mapl f
= foldr (\ a mbs → do{b ← f a; bs ← mbs; result (b:bs)})
     (result [])
```

Das Problem in der Funktion `mapl f` liegt darin, dass sie das Resultat von ihrem rekursiven Aufruf nicht an das `tail` von Liste anbindet, bevor sie das `head` von der Liste bearbeitet hat. Wenn die Argumentliste von `mapl` als eine snoc-Liste dargestellt und Funktion `mapl` durch `foldl` definiert wird, erst dann kann das Resultat vom rekursiven Aufruf angebunden werden.

```
mapl :: Monad m ⇒ (a → m b) → ([a] → m [b]=
mapl f
= foldl (\ mbs a → do{bs ← mbs; b ← f a; result (bs ++
         [b])}) (result [])
```

Dieses spezifische Muster von Listenfolding mit einer Monade, wird bei der *monadic fold left* auf Listen aufgegriffen:

```
mfoldl :: Monad m ⇒ (b → a → m b) → m b → ([a] → m b)
mfolfl msnoc mnil
= foldl (\ mb a →  do{b ← mb; msnoc b a}) mnil
```

Jetzt kann Funktion mapl als ein monadisches Fold definiert werden:

```
mapl f = mfoldl (\ bs a → do{b ← f a; result (bs ++ [b])})
(result []).
```

Als nächstes werden einige Beispiele von monadischer Listenfaltung aufgeführt. Die Funktion `accumulate` sammelt die Resultate aus der bearbeiteten Liste left-to-right.

```
accumalate :: Monad m ⇒ [m a] → m [a]
accumulate = foldr (\ ma mas → do{a ← ma;
                            as ← mas;
                            result (a:as)
                            }
                   )
                   (result [])
```

Mithilfe von Funktion `mfoldl` kann Funktion `accumulate` wie folgt definiert werden:

```
accumulate :: Monad m ⇒ [m a] → m [a]
accumulate = mfoldl (\ as ma → do{a ← ma;
                          result (as ++ [a])})(result [])
```

4.1 Substituting leaves revisited

Genau wie Funktion `mfoldr` auf Listen bearbeitet Funktion `mfoldB` auf die Monade durchgehend die rekursiven Teile des Typen.

```
mfoldB:: Monad m ⇒ (a → m b, b → b → m b) → (Tree a → m b)
mfoldB (mleaf, mnode)
= foldTree (\a → mleaf a, \mb  mb' →
                do{b ← mb; b' ← mb'; mnode b b'}
            )
```

Bemerkung: Evaluation der rekursiven Aufrufe zu Funktion `mfoldB` in der umgekehrten Reihenfolge resultiert in einer anderen Funktion.
Mithilfe vom monadischen Fold über Bäume `mfoldB` kann Funktion `subst` wie folgt definiert werden:

```
subst = mfoldB (\a → do{b ← lookup a; result (Leaf b)},
                \bs bs' → result (Node bs bs')
               )
```

Diese Definition ist deutlich kürzer und Übersichtlicher als die Definition derselben Funktion aus dem Abschnitt 4.2.:

```
subst :: Tree a → ReaderM [Assoc a b] (Tree b)
     subst
     = foldB (bindR (\ a → (lookup a)
                    (\ b → resultR (Leaf b)),
              bindR (\ mbs mbs' → mbs)
                    (bindR (\ bs → mbs')
                           (\ bs' → resultR (Node bs bs'))
                    )
             )
```

4.2 Umbenennen von λ-Ausdrücken

Weiter sind die monadischen Folds über Ausdrücke und Typen dargestellt.

```
mfoldT :: Monad m ⇒ (String → m a,a → a → m a) → Type
                  → m a
mfoldT (mtvar, marrow) = foldT (mtvar, \ms mt →
                                do{s ← ms; t ← mt;
                                   marrow s t}
                               )

mfoldE :: Monad m ⇒
          (String → m a,a → a → m a,)
           a → (String, Type) → a → m a) → (Expr → m a)
mfoldE (mvar, mapply, mlambda)
= foldE (\x         → mvar x,
         \mf ma     → do{f ← mf; a ← ma; mapplay f a},
        \(x,t) ma   → do{a ← ma; mlambda (x,t) a}
       )
```

Der Typechecker (|-) ist ein gutes Beispiel für eine Funktion, die als monadisches Fold nicht ausgedrückt werden kann. Das Problem liegt darin, dass der Körper einer λ-Abstraktion erst dann geprüft werden kann, wenn die Basis um einen zusätzlichen erwarteten Typ erweitert wurde. Ein Beispiel einer Funktion, die dem rekursiven Muster von Funktion `mfoldE` entspricht, ist die `rename` Funktion für λ- Ausdrücke von Wadler (1990). Diese Funktion benutzt die Zustandsmonade, um Quelle eines neuen Variablennamen einzugeben. Die Funktion ruft `substv x y b` auf und ersetzt alle freien Vorkommnissen von Variable `x` in `b` durch Variable `y`.

```
rename :: Expr → State Id String Expr
rename
= mfoldE (\x → result (Var x),\f a → result (f :@: a),
          \(x,t) ma → do{y ← new_name;
                         result ((x,t) ::: (substv x y b))})
```

5. Literaturverzeichnis

[1] Erik Meijer and Johan Jeuring. Merging Maps and Folds for Functional Programming. LNCS 925, 1995.

[2] Erik Meijer, Maarten Fokkinga, Ross Patterson: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, ACM Conference on Functional Programming and Computer Architecture 1991.

[3] Erik Meijer and Graham Hutton. Bananas in Space: Extending fold and unfold to Exponential Types. In Proc. ACM Conference on Functional Programming and Computer Architecture (FPCA) 1995.

[4] Prof. Dr. Rita Loogen. Skript zur Vorlesung Praktische Informatik III: Deklarative Programmierung. FB Mathematik und Informatik Philipps-Universität Marburg, WS 2001/02