

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

Monaden

<http://files.mehnle.net/studies/monaden.pdf>

Vortrag am 2000-02-01
Revision A vom 2001-10-10

Julian Mehnle
julian@mehnle.net

Inhaltsverzeichnis

0	Einführung	3
1	Varianten eines Auswerters	4
1.0	Ein einfacher Auswerter	4
1.1	Der Auswerter mit Exceptions.....	5
1.2	Der Auswerter mit globalem Zustand	6
2	Die Auswerter mit Monaden	6
2.0	Was eine Monade ausmacht	6
2.1	Der monadische Auswerter mit Exceptions	8
2.2	Der monadische Auswerter mit globalem Zustand	9
2.3	Die Gesetze der Monaden.....	10
3	Parsen mit Monaden	12
3.0	Listen als Monade	12
3.1	Einiges zu Parsern.....	13
3.2	Ein sequenzierender Parser	14
3.3	Ein alternierender Parser	16

0 Einführung

Um mathematische Beweise zu führen oder die Korrektheit von Algorithmen zu zeigen, eignen sich funktionale Programmiersprachen meist sehr gut. Allerdings wird dazu oft nur ein bestimmter Typ von Sprachen benutzt, die sogenannten "reinen" funktionalen Sprachen (von engl. "pure"), welche sich von den "unreinen" ("impure") funktionalen Sprachen dadurch unterscheiden, dass alle Berechnungen nur durch Funktionsanwendung durchgeführt werden, und dass sie einige Nebenwirkungen der unreinen Sprachen nicht zulassen. Diese Nebenwirkungen, wie z.B. implizite Zustandsänderungen oder Exception-Behandlung, erschweren im Allgemeinen eine Beweisführung recht erheblich. Leider ist es jedoch nicht selten der Fall, dass die Verwendung einer unreinen Sprache die Implementierung eines Algorithmus sehr vereinfachen, wenn nicht gar trivialisieren würde. Zu den reinen, funktionalen Sprachen zählen Miranda und das etwas besser bekannte Haskell. Unreine, funktionale Sprachen sind z.B. Scheme sowie Standard ML (SML).

Man hat lange nach möglichen Wegen gesucht, um die Vorzüge von unreinen Sprachen mit der leichten Beweisbarkeit in reinen Sprachen kombinieren zu können. In den 70ern kam man auf die Idee, das bereits einige Jahre zuvor in der Kategorientheorie erfundene Konzept der "Monaden" auf die funktionale Programmierung anzuwenden. Wie sich zeigte, ergab dies einige interessante neue Möglichkeiten, bisher als nur schwer oder umständlich lösbar betrachtete Probleme in der reinen, funktionalen Programmierung auf elegantem Wege zu lösen.

Monaden bieten auch Möglichkeit, Programme so zu strukturieren, dass sich viele Änderungen einfach durch Umdefinierung der verwendeten Monade und einigen geringfügigen Modifikationen am eigentlichen Programm erreichen lassen, was im Allgemeinen für eine hohe Übersichtlichkeit und leichte Wartbarkeit des desselben sorgt

Im Folgenden soll die Anwendung von Monaden in Haskell [3] gezeigt werden; dazu werden zunächst zwei Beispiele gegeben, die die Exception-Behandlung sowie die Verwaltung eines globalen Zustands unter der Verwendung von Monaden demonstrieren. Im Anschluss daran wird auf die formellen Eigenschaften von Monaden eingegangen, sowie auf die Gesetze, die für sie gelten. Zum Abschluss wird gezeigt, wie Monaden zur Erstellung von Parsern verwendet werden können. Parser verarbeiten eine Reihe von Eingabe-Daten (z.B. eine Formel aus Zahlen und Operatoren) in einen Ausgabe-Datum (z.B. das Ergebnis der Formel).

1 Varianten eines Auswerters

Ein Auswerter ist ein Programm, das einen Term, bestehend aus weiteren zusammengesetzten Termen oder aus atomaren Termen, auswertet, und anschliessend das Ergebnis zurückliefert. Obwohl das Konzept der Monaden bei Weitem nicht auf Auswerter beschränkt ist, soll es hier zur einfachen Demonstration zunächst bei einem solchen bleiben. Als Sprache kommt – wie oben bereits erwähnt – Haskell zum Einsatz.

1.0 Ein einfacher Auswerter

Als erstes gilt es, ein Grundgerüst für den Auswerter und zukünftige Modifikationen an ihm zu schaffen. Dazu definieren wir den Datentyp der Terme, die ausgewertet werden sollen:

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

Demnach ist ein Term entweder eine Konstante vom Typ Integer oder aber ein Quotient aus zwei weiteren Termen. Damit lassen sich Ausdrücke wie z.B. der folgende konstruieren:

```
(Div (Div (Con 1972) (Con 2)) (Con 23))
```

Der dazu passende Auswerter sieht dann so aus:

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

Nun definieren wir den obigen Beispiel-Term. Ausserdem wollen wir einen fehlerhaften Term definieren:

```
answer, error  :: Term
answer         = (Div (Div (Con 1972) (Con 2)) (Con 23))
error         = (Div (Con 1) (Con 0))
```

Der Term *answer* berechnet sich zu 42. Der Wert des Terms *error* ist undefiniert, da bislang keine Fehlerbehandlung angewendet wird.

1.1 Der Auswerter mit Exceptions

In unreinen Sprachen wird zur Fehlerbehandlung oft das Konzept von Exceptions (Ausnahmen) verwendet. Wird eine ungültige Situation erkannt, so wird sozusagen der "Ausnahmezustand" ausgerufen, auf dass die umschliessende Fehlerbehandlung diesen abfangen und das Problem bereinigen kann.

Wollen wir nun eine Fehlerprüfung in unseren Auswerter einbauen, so könnten wir z.B. eine neue Klasse von Funktionen einführen, bei deren Berechnung eine Exception auftreten kann:

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

Dabei sind *Raise* und *Return* Konstruktoren, die entweder eine Fehlermeldung vom Typ *String* oder ein Ergebnis eines beliebigen Typs *a* zurückliefern.

Jetzt können wir den Auswerter entsprechend umformen, um die Division durch 0 abzufangen und eine Exception zu erzeugen:

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

Falls bei der Berechnung des Divisors oder des Dividenden eine Exception auftritt, so wird diese erneut ausgerufen (mit dem selben Text), ansonsten wird der Quotient aus beiden berechnet. Falls hierbei der Divisor 0 ist, so wird ebenfalls eine Exception erzeugt, andernfalls wird schliesslich das Ergebnis der Division zurückgeliefert. Angewendet auf die beiden Beispiel-Terme liefert der Auswerter:

```
eval answer = (Return 42)
eval error  = (Raise "divide by zero")
```

1.2 Der Auswerter mit globalem Zustand

Angenommen wir wollen statt einer Fehlerprüfung jetzt die Anzahl der nötigen Divisionen berechnen. In einer unreinen Sprache würde man sinnvollerweise eine einfache, globale Zählervariable verwenden, die bei jeder Division um eins erhöht wird.

Da wir diese Möglichkeit nicht haben, verlegen wir uns erneut auf die Einführung eines neuen Typs von Funktionen, die neben ihrer normalen Funktion zusätzlich einen Zustand verändern:

```
type M a           = State → (a, State)
type State         = Int
```

Ein Wert vom Typ $M\ a$ sei nun eine Funktion, die den ursprünglichen Zustand entgegennimmt und das Ergebnis sowie den neuen Zustand zurückliefert. Der Typ $State$ sei hier zweckmässigerweise Int .

Sehen wir uns nun den umgeformten Auswerter an, so fällt auf, dass sich der Umfang der Änderungen gegenüber dem einfachen Auswerter in Grenzen hält:

```
eval                :: Term → M Int
eval (Con a) x      = (a,x)
eval (Div t u) x    = let (a,y) = eval t x in
                    let (b,z) = eval u y in
                    (a ÷ b, z + 1)
```

Bei jedem Aufruf des Auswerters muss der alte Zustand übergeben, der neue Zustand aus dem Ergebnis gewonnen und korrekt weitergegeben werden. Betrachten wir wieder den Term *answer* (der Term *error* erzeugt wie beim einfachen Auswerter *undefined*), so erhalten wir diesmal:

```
eval answer 0 = (42, 2)
```

2 Die Auswerter mit Monaden

2.0 Was eine Monade ausmacht

Anhand der Beispiele kann man vielleicht schon die Idee einer Monade erahnen: in jedem Fall haben wir zuerst einen neuen Typ eingeführt, der die jeweils gewünschte Form von Berechnung repräsentierte. Das M in den

verschiedenen Typen steht, wie nicht anders zu erwarten war, für *Monade*. Jede Funktion vom Typ $a \rightarrow b$ wurde durch eine entsprechende Funktion vom Typ $a \rightarrow M b$ ersetzt. M entspricht hier einer zusätzlich eingeführten Nebenwirkung, wie z.B. eben die Erzeugung von Exceptions, die Verwaltung eines globalen Zählers oder die Ausgabe von Text.

Welche Operationen jetzt müssen auf Typ M ausgeführt werden? Zum einen wäre da die Einheitsoperation, im Folgenden mit *unit* bezeichnet. *unit* überführt einen Wert vom Typ a in eine Funktion, die den Wert zurückliefert und sonst nichts besonderes tut:

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

Zum anderen brauchen wir eine Möglichkeit, eine Funktion vom Typ $a \rightarrow M b$ auf eine Berechnung vom Typ $M a$ anzuwenden. Praktischerweise schreibt man diese Operation als Infix-Funktion:

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

Eine Monade ist nun ein Tripel $(M, unit, \star)$, mit M als Typkonstruktor und den beiden Operationen *unit* und \star . (Diese beiden Operationen müssen ausserdem den in Abschnitt 2.3 gegebenen Gesetzen genügen.)

Im Folgenden werden oft Ausdrücke der Form

$$m \star \lambda a.n$$

auftreten, wobei m und n für Ausdrücke sind, und a eine Variable ist. Die Form $\lambda a.n$ ist ein Lambda-Ausdruck, mit n als Gültigkeitsbereich von a . Lesen lässt sich der Ausdruck wie folgt: führe Berechnung m aus, binde a an das Ergebnis und führe dann Berechnung n aus. Sieht man sich die Definition von \star an, so stellt man fest, dass der Ausdruck m vom Typ $M a$ ist, die Variable a vom Typ a , der Ausdruck n vom Typ $M b$, der Lambda-Ausdruck $\lambda a.n$ vom Typ $a \rightarrow M b$, und der gesamte Ausdruck ist vom Typ $M b$.

Anhand dieser Abstraktionen lässt sich der Auswerter wie folgt umschreiben:

$$\begin{aligned} eval & :: Term \rightarrow M Int \\ eval (Con a) & = unit a \\ eval (Div t u) & = eval t \star \lambda a. eval u \star \lambda b. unit (a \div b) \end{aligned}$$

Anzumerken ist hier, dass die Lambda-Abstraktion am schwächsten bindet, und Funktionsanwendung am stärksten. Der Auswerter liest sich dann so: Um $(Con\ a)$ zu berechnen, gib einfach a zurück. Um $(Div\ t\ u)$ zu berechnen, berechne zuerst t , binde a an das Ergebnis, berechne dann u , binde b an das Ergebnis, und gib schliesslich $a \div b$ zurück.

```

type  $M\ a$            =  $a$ 

   $unit$                 ::  $a \rightarrow M\ a$ 
   $unit\ a$              =  $a$ 

  ( $\star$ )              ::  $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$ 
   $a \star k$            =  $k\ a$ 
  
```

Nimmt man diese Definitionen, welche übrigens die sog. Einheitsmonade darstellen, zum obigen Auswerter hinzu, so lässt sich durch Einsetzen und Vereinfachen exakt der einfache Auswerter aus Abschnitt 1.0 ableiten.

2.1 Der monadische Auswerter mit Exceptions

Definieren wir zunächst die Exception-Monade:

```

data  $M\ a$            =  $Raise\ Exception \mid Return\ a$ 
type  $Exception$     =  $String$ 

   $unit$                 ::  $a \rightarrow M\ a$ 
   $unit\ a$              =  $Return\ a$ 

  ( $\star$ )              ::  $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$ 
   $m \star k$            = case  $m$  of
                         $Raise\ e \rightarrow Raise\ e$ 
                         $Return\ a \rightarrow k\ a$ 

   $raise$                ::  $Exception \rightarrow M\ a$ 
   $raise\ e$            =  $Raise\ e$ 
  
```

Um eine Fehlerbehandlung zum monadischen Auswerter hinzuzufügen, ersetzen wir $unit (a \div b)$ durch:

```

if  $b == 0$ 
  then raise "divide by zero"
  else  $unit (a \div b)$ 

```

Der Änderungsaufwand entspricht durchaus dem in einer unreinen Sprache. Wiederum ist der Auswerter vollkommen gleichwertig zu dem in Abschnitt 1.1.

2.2 Der monadische Auswerter mit globalem Zustand

Die Zustands-Monade sieht folgendermassen aus:

```

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

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

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

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

```

Der Aufruf $unit a$ liefert die Berechnung zurück, die den alten Zustand x übernimmt und den Wert a und den neuen Zustand x zurückgibt, d.h. der Zustand bleibt unverändert. Der Aufruf $m \star k$ führt im alten Zustand x die Berechnung m aus, welche den Wert a und den zwischenzeitigen Zustand y liefert, und führt dann die Berechnung $k a$ in Zustand y aus, woraus das Endergebnis b sowie der Endzustand z resultieren. Der Aufruf $tick$ erhöht den Zustandszähler und liefert den leeren Wert $()$ zurück, dessen Typ ebenfalls als $()$ geschrieben wird.

Damit der monadische Auswerter die durchgeführten Berechnungen zählt, ersetzen wir diesmal $unit(a \div b)$ durch:

$$tick \star \lambda(). unit(a \div b)$$

Wieder ähneln die Änderungen denjenigen, die in einer unreinen Sprache vonnöten wären. Erwartungsgemäss erhält man durch Einsetzen der Definitionen von \star und $unit$ und Vereinfachen den Auswerter mit globalem Zustand aus Abschnitt 1.2.

2.3 Die Gesetze der Monaden

Es gibt drei Gesetze, die die beiden Monaden-Operationen erfüllen müssen:

- *Links-Einheit.* Berechne den Wert a , binde b an das Ergebnis, und berechne n . Das Ergebnis ist dasselbe wie n mit der Variablen b ersetzt durch den Wert a .

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

- *Rechts-Einheit.* Berechne m , binde a an das Ergebnis, und gib a zurück. Das Ergebnis ist dasselbe wie m .

$$m \star \lambda a. unit a = m$$

- *Assoziativität.* Berechne m , binde a an das Ergebnis, berechne n , binde b an das Ergebnis, berechne o . Die Anordnung der Klammern in solch einer Berechnung ist gleichgültig.

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

Der Gültigkeitsbereich der Variablen a beinhaltet o auf der linken Seite, schliesst o auf der rechten Seite jedoch aus. Somit gilt dieses Gesetz nur, wenn a nicht frei in o vorkommt.

Ergänzend sollen hier noch die zwei Funktionen map und $join$ vorgestellt werden, die einige recht nützliche Anwendungen besitzen und ausserdem eine alternative Möglichkeit darstellen, eine Monade zu definieren:

$$\begin{aligned}
 \text{map} & & & :: (a \rightarrow b) \rightarrow (M a \rightarrow M b) \\
 \text{map } f \ m & & & = m \star \lambda a. \text{unit } (f a) \\
 \\
 \text{join} & & & :: M (M a) \rightarrow M a \\
 \text{join } z & & & = z \star \lambda m. m
 \end{aligned}$$

Die Operation *map* überführt eine Funktion *f* des Typs $a \rightarrow b$ in eine monadische Berechnung (mit den Nebenwirkungen der durch \star und *unit* definierten Monade) des Typs $M a \rightarrow M b$. Die Operation *join* verbindet zwei Berechnungen bzw. deren Nebenwirkungen zu einer gemeinsamen.

Man kann *map* auch dazu verwenden, eine Funktion auf jedes Element einer Liste – wie später gezeigt wird, stellen auch Listen eine Monade dar – anzuwenden, und *join* dazu, eine Liste aus Listen zu erstellen!

Mittels *map* und *join* kann man weitere Gesetze aufstellen. Dabei sei *id* die Einheitsfunktion ($id\ x = x$) und (\cdot) die Komposition zweier Funktionen ($(f \cdot g)\ x = f(g\ x)$):

$$\begin{aligned}
 \text{map } id & & & = id \\
 \text{map } (f \cdot g) & & & = \text{map } f \cdot \text{map } g \\
 \\
 \text{map } f \cdot \text{unit} & & & = \text{unit} \cdot f \\
 \text{map } f \cdot \text{join} & & & = \text{join} \cdot \text{map } (\text{map } f) \\
 \\
 \text{join} \cdot \text{unit} & & & = id \\
 \text{join} \cdot \text{map } unit & & & = id \\
 \text{join} \cdot \text{map } join & & & = \text{join} \cdot \text{join} \\
 \\
 m \star k & & & = \text{join } (\text{map } k\ m)
 \end{aligned}$$

Der Beweis dieser Gesetze folgt aus den Definitionen von *map* und *join* sowie den drei Monaden-Gesetzen. Man kann Monaden statt über *unit* und \star auch über *unit*, *join* und *map* definieren [4, 5]. Dann treten die ersten sieben der obigen Gesetze an die Stelle der drei zuvor genannten Monaden-Gesetze. Beide Definitionsweisen sind absolut gleichwertig.

3 Parsen mit Monaden

Parser (von engl. to parse, für: zerlegen, analysieren, (zer)gliedern) werden bei vielen Gelegenheiten verwendet, z.B. in Compilern, die aus dem Quellcode eines Programms entweder eine Art Meta-Code oder gar einen direkt ausführbaren Maschinencode generieren, oder in Grammatik-Prüfungssystemen von Textverarbeitungsprogrammen. Aber natürlich auch für die Verarbeitung oder Überprüfung von BNF-Grammatiken können Parser verwendet werden.

In diesem Abschnitt wird zunächst erklärt, wie sich Listen als Monade auffassen lassen, da diese eine wichtige Grundlage für Parser darstellen. Dann sollen zwei Beispiele zeigen, wie leicht sich mit Hilfe von Monaden einfache Parser konstruieren lassen, nämlich ein sequenzierender und ein alternierender Parser.

3.0 Listen als Monade

Eine Liste ist eine Aneinanderreihung von Elementen des Basistyps der Liste. Ist a der Basistyp, so ist der Typ der Liste $[a]$. Die Konkatenation von Elementen geschieht mit $:$, und das Ende einer Liste bildet immer die leere Liste. Somit wäre $[1, 2, 3] = 1 : 2 : 3 : []$ eine Liste des Typs $[Int]$ mit Elementen des Typs Int . *String* ist übrigens nur ein Synonym für $[Char]$, es gilt: "*Monade*" = $['M', 'o', 'n', 'a', 'd', 'e']$.

Vereinbaren wir nun die folgende Definition:

$$\begin{aligned}
 unit & & & :: a \rightarrow [a] \\
 unit\ a & & & = [a] \\
 \\
 (\star) & & & :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b] \\
 [] \star k & & & = [] \\
 (a : x) \star k & & & = k\ a ++ (x \star k)
 \end{aligned}$$

unit erzeugt die Liste mit a als einzigem Element als Inhalt. $m \star k$ wendet k auf jedes Element der Liste m an und hängt die resultierenden Listen aneinander.

Im Folgenden wird auf die sogenannte "List-Comprehension"-Notation zurückgegriffen. Diese Notation erlaubt eine sehr einfache Erzeugung und Verarbeitung von Listen. Sie ähnelt der Schreibweise, die in der Mathematik oft für Mengen verwendet wird. Beispiele wären:

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

Wie man sieht, erzeugt das zweite Konstrukt eine Liste bestehend aus Tupeln mit allen möglichen Kombinationen der Elemente aus den beiden angegebenen Listen $[1, 2]$ und "list" .

Die List-Comprehension-Notation lässt sich wie folgt in Monaden-Operationen übersetzen:

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

t ist hier ein Ausdruck, x und y sind Variablen (oder allgemeiner: Muster (engl. Patterns)), und u und v sind Ausdrücke, die sich zu Listen berechnen. Mehr über die Verbindungen zwischen Comprehensions und Monaden ist in [2] nachzulesen.

3.1 Einiges zu Parsern

Parser lassen sich wie folgt darstellen:

$$\begin{aligned} \text{type } M a &= \text{State} \rightarrow [(a, \text{State})] \\ \text{type } \text{State} &= \text{String} \end{aligned}$$

Ein Parser für den Typ a nimmt einen Zustand in Form des zu parsenden Strings entgegen und liefert eine Liste zurück mit dem vom String geparsten Wert a , sowie einem Zustand, der den verbleibenden, noch zu parsenden String repräsentiert. Die Liste enthält *alle* Möglichkeiten, den Urzustand zu parsen: sie ist leer, falls sich der Zustand nicht parsen lässt, sie enthält ein Element, falls sich der Zustand auf eindeutige Weise parsen lässt, zwei Elemente bei zwei Möglichkeiten und so weiter.

Betrachten wir einen einfachen Parser für arithmetische Ausdrücke, der einen Baum des bereits zuvor verwendeten Typs zurückliefert:

data $Term = Con\ Int \mid Div\ Term\ Term$

Angenommen, wir haben einen entsprechenden Parser:

$term :: M\ Term$

Dann sind dies einige Beispiele für seine Anwendung:

$term\ "23"$ = $[(Con\ 23, "")]$
 $term\ "23\ and\ more"$ = $[(Con\ 23, " and more")]$
 $term\ "not\ a\ term"$ = $[]$
 $term\ "((1972\div 2)\div 23)"$ = $[(Div\ (Div\ (Con\ 1972)\ (Con\ 2))\ (Con\ 23)), ""]]$

Ein Parser heisst *eindeutig*, wenn die Liste aller Möglichkeiten $x\ m$, die Eingabe x zu parsen, für alle x entweder leer ist, oder genau ein Element enthält. Zum Beispiel ist der obige Parser $term$ eindeutig. Ein *mehrdeutiger* Parser kann mehrere alternative Möglichkeiten zurückliefern, die Eingabe zu parsen.

Der einfache Parser liefert immer das erste Element der Eingabe, es sei denn die Eingabe ist leer bzw. vollständig verarbeitet:

$item$ $:: M\ Char$
 $item\ []$ = $[]$
 $item\ (a : x)$ = $[(a, x)]$

Zwei Beispiele dazu:

$item\ ""$ = $[]$
 $item\ "Monade"$ = $[('M', "onade")]$

Ohne Zweifel ist der Parser $item$ eindeutig.

3.2 Ein sequenzierender Parser

Nun wollen wir einen sequenzierenden Parser betrachten. Dieser Parser soll immer ganze Sequenzen aus der Eingabe parsen, in diesem Falle genau zwei Zeichen.

Zunächst benötigt man aber speziell auf Parser ausgerichtete Definitionen für *unit* und \star :

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

Der Parser *unit a* nimmt die Eingabe *x* und liefert den geparsen Wert *a* und die verbleibende Eingabe *x* zurück. Der Parser *m \star k* nimmt die Eingabe *x* entgegen; der Parser *m* wird auf die Eingabe *x* angewendet, was für jeden möglichen Parse einen Wert *a* sowie die verbleibende Eingabe *y* ergibt; dann wird der Parser *k a* auf diese verbleibende Eingabe *y* angewendet, und schliesslich erhält man für jeden möglichen Parse einen Wert *b* sowie die insgesamt verbleibende Eingabe *z*.

Somit entspricht *unit* dem "leeren" Parser, der keine Eingabe verbraucht, und \star entspricht dem sequenzieren von Parsern.

Man kann nun auf folgende Weise zwei Elemente parsen:

$$\begin{aligned}
 \textit{twoItems} & && :: M (\textit{Char}, \textit{Char}) \\
 \textit{twoItems} & && = \textit{item} \star \lambda a. \textit{item} \star \lambda b. \textit{unit} (a, b)
 \end{aligned}$$

Zum Beispiel:

$$\begin{aligned}
 \textit{twoItems} \ "m" & && = [] \\
 \textit{twoItems} \ "Monade" & && = [((\textit{'M'}, \textit{'o'}), \textit{"nade"})]
 \end{aligned}$$

Das Parsen ist nur erfolgreich, solange die Liste (noch) mindestens zwei Elemente enthält.

3.3 Ein alternierender Parser

Abschliessend soll noch ein alternierender Parser untersucht werden. Ein solcher Parser liefert nicht – wie der sequenzierende Parser – nur einen Parse (im obigen Fall zwei Elemente) zurück, sondern versucht, mehrere – alternative – Parses durchzuführen. Dazu definieren wir uns zwei neue Funktionen:

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

Der Parser *zero* nimmt die Eingabe *x* entgegen, versagt jedoch immer. Der Parser $m \oplus n$ nimmt die Eingabe *x* entgegen und liefert alle möglichen Parses von *m* angewendet auf die Eingabe *x* sowie alle möglichen Parses von *n* angewendet auf die selbe Eingabe *x* zurück.

Dies nun ist der Parser, der entweder ein oder zwei Elemente der Eingabe parst:

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

Erneut einige Beispiele zu diesem Parser:

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

Das letzte Beispiel liefert zwei alternative Parses. Dies zeigt, dass man durch Alternierung mehrdeutige Parser erhalten kann.

Literaturverzeichnis

Diese Arbeit ist lehnt sich im Allgemeinen eng an [1] an.

- 1 [Philip Wadler](#), University of Glasgow,
Monads for functional programming, erschienen in:
Advanced Functional Programming,
Springer, Mai 1995
<http://www.cs.bell-labs.com/~wadler/topics/monads.html#marktoberdorf>
- 2 [Philip Wadler](#), University of Glasgow,
Comprehending Monads, erschienen in:
Conference on Lisp and Functional Programming,
ACM, Juni 1990
<http://www.cs.bell-labs.com/~wadler/topics/monads.html#monads>
- 3 P. Hudak, S. Peyton Jones, P. Wadler,
Report on the Programming Language Haskell: Version 1.1,
Yale Universität, Glasgow Universität, August 1991
- 4 S. Mac Lane,
Categories for the Working Mathematician,
Springer, 1971
- 5 E. Moggi,
Computational lambda-calculus and monads, erschienen in:
Symposium on Logic in Computer Science,
IEEE, Juni 1989
- 6 [Graham Hutton](#)
Frequently Asked Questions for `comp.lang.functional`
<http://www.cs.nott.ac.uk/~gmh/faq.html>