Die funktionale Programmiersprache Haskell

Typklassen und Monaden

Lars Hupel

28. Juni 2009

1 Einführung in Haskell

Haskell heißt die 1990 veröffentlichte und nach dem US-amerikanischen Mathematiker Haskell B. Curry benannte funktionale Programmiersprache, die sich unter anderem durch ihr reiches Typsystem auszeichnet – sie folgt dem typisierten Lambda-Kalkül und ist daher auch in der Lage, Typen zu inferieren. Für die Benennung eines weiteren Konzepts in Haskell stand ebenfalls Curry Pate, nämlich *Currying* bzw. (selten) *Schönfinkeln*, was die Behandlung von Funktionen als Werte erheblich vereinfacht. Diese und andere Eigenschaften sollen im Folgenden erklärt werden.

1.1 Syntax

Haskells Syntax besteht grundlegend aus den Definitionen von Termen und der Typangabe:

```
x :: Int
x = 3

f :: Int -> Int
f y = x + y
```

Der erste Teil definiert eine Variable `x` vom Typ `Int`; der zweite Teil eine Funktion `f`, deren Urbilder und Bilder beide vom Typ `Int` sind. Jeder Ausdruck kann durch Nachstellen von `::` und dem Typ explizit typisiert werden, was

vor allem dann nützlich ist, wenn man nicht den generischen Typen erhalten möchte, sondern einen speziellen (im obigen Beispiel wäre statt `Int` auch noch z. B. `Float` möglich).

Zusätzlich gibt es die gängigen Konstrukte aus anderen Programmiersprachen, z. B. Listen und Tupel:

```
[1,2,3,4] :: [Int]
(1,2,2.5) :: (Int,Int,Float)
```

Eine Besonderheit von Haskell sind dabei die Kurzschreibweisen für Listen, genannt *arithmetische Listen* bzw. *Listenkomprehension*:

```
list = [1,2..4]

let even n = (n `mod` 2) == 0
in [ x | x <- list, even x ]
```

Die erste Anweisung erzeugt eine arithmetische Liste, die mit 1 beginnt und anschließend alle ganzen Zahlen bis 4 enthält. Der Ausdruck `let` ... `in` ... bindet einen lokalen Bezeichner im folgenden Ausdruck (fast identisch zu ...`where`...). Anschließend wird per Listenkomprehension die Liste aller `x` gebildet, die aus `list` stammen und die Bedingung `even n` erfüllen. Das Resultat ist wie erwartet `[2,4]`.

Man kann in Haskell Funktionen als Operator nutzen, in dem man diese in Backticks (`) schreibt, wie oben bei `mod`. Umgekehrt kann man auch Operatoren durch Einklammern als Funktionen verwenden, wodurch

```
(+) x y = x - y
```

eine gültige Definition in Haskell darstellt.

Mittels des ...`where`...-Konstrukts können Werte an Variablen "zugewiesen" werden. Diese sind aber, wie in allen funktionalen Sprachen, nicht mehr veränderbar. Nützlich ist diese Funktionalität, wenn man längere Ausdrücke abkürzen möchte.

```
listMax :: [Int] -> Int
listMax [] = error "Kann das Maximum einer leeren Liste nicht
   bestimmen"
listMax (x : xs) = maxl x xs where
  maxl m [] = m
  maxl m (h : t) = if m > h
                     then (maxl m t)
                     else (maxl h t)
```

Hier wird eine Funktion `listMax` definiert, die das Maximum einer Liste von `Int`s liefern soll. Bei einer leeren Liste soll die Auswertung abbrechen. Wie andere funktionale Programmiersprachen unterstützt auch Haskell *Musterabgleich* bzw. *Pattern Matching*, hier mit `x : xs`, was eine Liste in Kopf und Rumpf aufteilt (insbesondere ist `[1,2,3,4]` eine Kurzform von `1:2:3:4:[]`).

Wie bereits oben angesprochen, unterstützt Haskell Currying. Damit wird eine Technik bezeichnet, die das *partielle* Anwenden einer Funktion auf einen Teil der Parameter erlaubt. Konkret bedeutet das, dass – im Gegensatz zu imperativen Programmiersprachen – Parameter nicht als Tupel erwartet werden. Das partielle Anwenden lässt sich damit am ehesten mit dem "Binden" eines Parameters an eine Funktion umschreiben, was eine neue Funktion als Resultat hat. Eine nützliche Anwendung dafür stellen dabei die *Sektionen* dar:

```
map (+4) [1,2,3]
```

Die Sektion `(+4)` bindet dabei von rechts den Parameter `4` an den Operator `+`, wodurch eine Funktion "vier addieren" mit nur noch einem Parameter gebildet wird. Dieser Code bildet folglich die Liste `[1,2,3]` auf die Liste `[5,6,7]` ab, was mittels `map` umgesetzt wird. Im Allgemeinen (d. h. bei beliebigen Funktionen) lassen sich allerdings Parameter nur von links binden:

```
g :: Int -> Int
g = min 10
```

– diese Funktion wendet `min` partiell auf `10` an und bestimmt daher das Minimum von 10 und einem angegebenen Parameter.

Fast alle Funktionen mit mehreren Paramtern werden "gecurryt" angegeben, um eine partielle Anwendung zu erlauben. So schriebe man, um nocheinmal auf das obige Beispiel von `listMax` zurückzukommen, für den Typen von der "inneren" Funktion `maxl`

```
maxl :: Int -> [Int] -> Int
```

statt

```
maxl :: (Int,[Int]) -> Int
```

Auf einige weitere Syntaxkonstrukte wird später noch eingegangen.

1.2 Datentypen

Eigene Datentypen können in Haskell über eine Menge von Konstruktoren definiert werden:

```
data Shape = Rectangle Float Float
           | Ellipse Float Float
           | Triangle Float Float
           | Polygon [(Float,Float)]
```

Dabei wird der Datentyp `Shape` deklariert, der über vier Konstruktoren verfügt.

Synonyme werden mit dem Schlüsselwort `type` eingeführt:

```
type Side = Float
type Radius = Float
type Vertex = (Float,Float)
```

Oftmals stellt man verschiedene Sachverhalte mittels der gleichen Repräsentation dar; so sind z. B. "Seitenlänge" und "Radius" verschiedene "Objekte", nutzen aber beide ein `Float` als Darstellung. Mit der obigen Definition sind Seitenlängen und Radien aber beliebig austauschbar, was unerwünscht ist. Einen eigenen Typen mit Konstruktoren zu definieren erscheint hier praktisch, ist aber mit zusätzlichem Aufwand beim Musterabgleich verbunden. Für diese Fälle gibt es ein weiteres Schlüsselwort:

```
newtype Point = Point (Float,Float)
```

Nun hat der Typ `Point` einen einzigen Konstruktor, nämlich `Point` (trotz selben Namens handelt es sich um verschiedene Objekte, nämlich Typ und Konstruktor). Da bei `newtype` grundsätzlich nur ein Konstruktor erlaubt ist, wird eine Haskell-Implementation dies optimieren können.

Für die folgenden Ausführungen soll uns ein anschauliches Beispiel dienen, nämlich Sammlungen- bzw. Container-Typen. So könnte man Datentypen wie "Liste", "Baum", "Warteschlange", "Stapel" etc. definieren und die Handhabung derer über der Struktur abstrahieren. Wichtig ist, dass man *generische* Typen erhalten möchte, also solche, die von einem weiteren Typen abhängig sind. So möchte man also z. B. Listen von `Int`s oder Listen von Listen von `Float`s definieren. Solche Parameter werden in Haskell mit Bezeichnern, die mit kleinem Buchstaben beginnen benannt. Ein Beispiel aus der Standardbibliothek ist der oftmals verwendeter Datentyp `Maybe a`:

```
data Maybe a = Nothing | Just a
```

Dies entspricht der Umsetzung von "Nullpointern" in Haskell.

Listen und Bäume kann man daher wie folgt definieren:

```
newtype List a = List [a]
data Tree a = Empty | Branch (Tree a) a (Tree a)
```

2 Typklassen

Haskell definiert ein *Typklassen* genanntes Sprachmittel, welches grob mit *Vererbung* in gängigen objektorientierten Sprachen vergleichbar ist. Eine Typklasse definiert Funktionen mit fixer Signatur, die auf jede *Instanz* (bzw., in korrekter Übersetzung, *Exemplar* oder *Ausprägung*) anwendbar sein müssen.

2.1 Deklaration von Typklassen und Ausprägungen

Eine der "Standard"-, d. h. eingebauten, Typklassen stellt `Eq` dar:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

(Zur Erinnerung: Die Syntax `(==)` steht für den Operator `==`, der als Funktion behandelt wird und `::` für die Typannotation.)

Folglich muss jede Ausprägung on `Eq` diese beiden Operationen mitbringen:

```
instance Eq (Float,Float) where
  (x,y) == (u,v) = x == u && y == v
  v1 /= v2 = not (v1 == v2)
```

Die Äquivalenz ist hier natürlich trivial. Denkbar sind aber auch komplexere Definitionen, z. B. Äquivalenz modulo einer bestimmten Zahl.

Möglich ist aber auch die Angabe von Bedingungen bei der Definition einer Ausprägung:

```
instance Eq a => Eq [a] where
  [] == [] = True
  [] == _ = False
  (x : xs) == (y : ys) | x == y     = xs == ys
                       | otherwise = False
```

Die Syntax `Eq a =>`[1] (man beachte den Doppelpfeil) verlangt das gewünschte, nämlich dass `a` der Klasse `Eq` angehört. Die letzten beiden Zeilen nutzen die auch aus anderen Sprachen bekannten *Wächter*. Listen von `a` sind demzufolge vergleichbar, wenn Werte von `a` vergleichbar sind. Demzufolge sind auch Listen vorstellbar, die nicht vergleichbar sind (z. B. Listen von Funktionen); so etwas ist in den gängigen imperativen Sprachen nicht möglich.

[1] Die Angabe von Bedingungen ist nicht nur bei Ausprägungen möglich, sondern auch bei der Definition von Funktionen, z. B. `pow :: Num a =>a -> a -> a`.

2.2 Vererbung

Ein ähnlicher Mechanismus kommt auch bei Typklassen zur Verwendung:

```
class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
```

`Ord` erweitert (bzw. "erbt") die Operationen von `Eq` und fügt noch vier weitere hinzu. Ausprägungen von `Ord` müssen folglich auch Ausprägungen von `Eq` sein. Dieses Konzept entspricht genau der vererbungsbasierten Objektorientierung mit Hierarchien von (abstrakten) Klassen (nicht Schnittstellen, da in Typklassen auch "default"-Methoden angegeben werden können). Im Gegensatz zu Java, allerdings wie in C++ sind in Haskell auch Mehrfachvererbungen zulässig.

Möglich ist deshalb z. B. folgende Definition von `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Dies sieht auf dem ersten Blick nach einer Endlosrekursion aus; tatsächlich muss sich jedoch die Ausprägung selbst darum kümmern, dass mindestens eine der beiden Operationen "überschrieben" wird.

2.3 Beispiel: Sammlungen

Typklassen erlauben es nicht nur, mit einfachen Typen umzugehen, sondern auch mit sogenannten *Typkonstruktoren* – nicht zu verwechseln mit Konstruktoren –. Dabei handelt es sich um generische Typen wie `Tree`. Eine Typklasse für Sammlungen wie oben beschrieben, könnte deshalb wie folgt definiert sein:

```
class Collection c where
  add, remove :: c a -> a -> c a
  construct :: [a] -> c a
```

Der übergebene Typkonstruktor `c` wird dabei auf `a` angewendet; aus einem generischen Typen wird ein spezieller Typ gebildet. Folglich muss es sich bei `c` um einen "Typen mit Parameter" handeln. Wichtig ist dabei, diese Applikation eines Typs auf einen anderen nicht mit Funktionsapplikation zu verwechseln.

Ein Standardbeispiel ist hierbei `[a]`, was nichts anderes bedeutet, als den eingebauten Typen `[]` mit dem Parameter `a` zu verwenden.

Die Definition der Ausprägung für `List` ist nun offensichtlich[2]:

[2] ++ steht für Listenkonkatenation

```
instance Collection List where
  add (List list) elem = List (list ++ [elem])
  remove (List list) elem =
    List (aux list elem) where
      aux [] elem = []
      aux (x : xs) elem | x == elem = xs
                        | otherwise = x : (aux xs elem)
  construct xs = List xs
```

Allerdings gibt es hier ein Problem: Der Haskell-Interpreter wird diesen Code nicht ausführen, da unbekannt ist, ob der Vergleich `x == elem` überhaupt möglich ist. An keiner Stelle wurde gefordert, dass der Typ der Listenelemente eine Ausprägung von `Eq` ist. Es gibt hierfür mehrere Lösungsansätze:

- `Eq a` als Voraussetzung bei der Ausprägung von `remove` hinzufügen – nicht möglich, da dies elementare Gesetze der Vererbung verletzen würde (nicht kontravariant)
- `Eq a` als Voraussetzung bei der Typklasse angeben (`class Eq a =>Collection c where`...) – ebenfalls nicht möglich, da in der "Prämisse" eine Typvariable verwendet wird, die in der "Konklusion" nicht auftaucht
- `Eq a` als Voraussetzung bei `remove` in `Collection` hinzufügen – löst dieses Problem

Analog kann man mit der Ausprägung von `Tree` verfahren:

```
instance Collection Tree where
  add Empty elem = Branch Empty elem Empty
  -- usw.
  remove Empty _ = Empty
  -- usw.
  construct [] = Empty
  construct (x : xs) = add (construct xs) x
```

Hierfür braucht man allerdings, möchte man die Suchbaumeigenschaft nutzen, zusätzlich die Bedingung `Ord a`. Problematisch ist dabei, dass man an sich nicht die Definition der Typklasse ändern kann bzw. sollte, um eine neue Instanz zu definieren.

Dieses Problem ist allgemein bekannt und im derzeitigen Haskell-Standard noch nicht behoben [4]. Nichtsdestotrotz wurde in den gängigen Interpretern, z. B. dem *Glasgow Haskell Compiler* (*GHC*) das Typklassenkonzept erweitert, so dass u. a. Typklassen mehrere Argumente erwarten können; dies zu behandeln spränge jedoch den Rahmen dieser Arbeit.

2.4 Iteration über Elemente

Für allgemeine Sammlungen ist es oft sinnvoll, sogenannte *Iteratoren* bereitzustellen, wenngleich diese Bezeichnung bei einer funktionalen Programmiersprache etwas irreführend ist. Ein Iterator soll ermöglichen, dass man eine Art "listenbasierten" Zugriff auf Elemente hat, ohne tatsächlich eine Liste verwenden zu müssen. Analog zu den Möglichkeiten einer Liste kann man daher definieren:

```
class Iterator i where
  getElement :: i a -> a
  next :: i a -> i a
  isEmpty :: i a -> Bool
```

An dieser Klasse wird der Unterschied zwischen *Polymorphismus* und Typklassen deutlich:

> "Polymorphism captures similar structure over different values, while type classes capture similar operations over different structures" [1, Seite 152]

Der Vorteil von `Iterator` ist nun, dass man beliebige Algorithmen auf "Listen" ohne Kenntnis der tatsächlichen Struktur verwenden kann. Als Beispiel soll hier `foreach` dienen, eine gängige Kontrollstruktur. Der Algorithmus dafür soll eine Funktion auf alle Elemente der Sammlung anwenden und die Ergebnisse akkumulieren. Eine geeignete Definition sieht also wie folgt aus:

```
foreach :: Iterator i => i a -> (a -> b) -> (b -> c -> c) -> c -> c
foreach iter func acc start
  | isEmpty iter = start
  | otherwise    = (func (getElement iter)) `acc`
                   (foreach (next iter) func acc start)
```

3 Anwendungsbeispiel: Monaden

Bei funktionalen Programmiersprachen stellt sich stets die Frage, wie Benutzerinteraktion gesteuert wird. Es gibt mehrere Konzepte, wie Ein- und Ausgabe umgesetzt werden können, dabei ist aber Wert darauf zu legen, dass die funktionalen Konzepte nicht verletzt werden.

Angenommen, es gäbe eine Funktion `getInt :: Int`, bei der, wenn sie von einem geeigneten Interpreter aufgerufen wird, eine Zahl von der Standardeingabe gelesen würde. Das Resultat könnte man ebenso einfach mit `putInt :: Int -> ()`[3] schreiben. Die zweite Funktion ließe sich umsetzen, die erste hingegen ist

[3] `()` steht für den Typ der nullelementigen Tupel mit genau einem Wert, nämlich `()`

problematisch. Das Problem daran lässt sich an folgendem Ausdruck demonstrieren:

```
diff = getInt - getInt
```

Zunächst einmal ist das Resultat von `diff` abhängig von der Auswertungsreihenfolge, die allerdings bei den Parametern einer Subtraktion keine Rolle spielt. Möglicherweise könnten also verschiedene Haskell-Implementationen verschiedene Ergebnisse liefern. Das zweite Problem ist, dass das Schließen von Eigenschaften eines Programms unmöglich wird, da elementare Regeln wie `x - x == 0` nicht mehr gelten – es gibt *Seiteneffekte*.

Andererseits möchte man natürlich nicht auf Ein- und Ausgabe verzichten. Um diese Funktionalität ohne die genannten Einschränkungen trotzdem nutzen zu können, gibt es in Haskell das Konzept der *Monaden*. Dazu passend gibt es einerseits den Datentypen `IO a` (den man sich als "Aktion, die ein `a` als Resultat hat" vorstellen kann) und eine Typklasse `Monad m` (die eine Art "Verkettung" von Aktionen ermöglicht).

3.1 Umsetzung in Haskell

Folgt man der Analogie zu den imperativen Programmiersprachen, so ist eine Aktion eine *Zustandsänderung*, also eine Funktion, die einen Ursprungszustand in einen neuen Zustand überführt und dabei ein Resultat liefert; quasi ein "Funktionsaufruf mit Referenzparametern". Der Typ `IO` repräsentiert dabei den Zustand des Betriebssystems.

Ein- und Ausgabefunktionen sind daher wie folgt definiert:

```
putStr :: String -> IO ()
getLine :: IO String
```

Da man aber nicht selbst auf den Wert von `IO String` zugreifen kann, muss man sich der *Komposition* bedienen:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Diese Operation erhält zwei Argumente; das erste wird dabei "ausgeführt" und das Ergebnis als Argument für die übergebene Funktion verwendet. Da diese Funktion selbst auch wieder eine Monade liefert, ist das Ergebnis der gesamten Komposition wieder eine Monade.

Das Kopieren einer Zeile von der Eingabe in die Ausgabe funktioniert deshalb einfach mit

```
getLine >>= \x -> putStr x
```

Mittels `\x -> ...` wird hier eine Lambda-Funktion definiert. Statt $\lambda x.T$ schreibt man in Haskell `\x -> T`.

Zunächst wird die Aktion "Lesen einer Zeile von der Standardeingabe" getätigt und dessen Resultat an die Aktion "Schreibe eine Zeichenkette auf die Standardausgabe" weitergereicht. Die tatsächliche Ausführung dieser Anweisung, die selbst wieder eine Aktion ist, obliegt nun dem Interpreter, der eine Abbildung auf die Systemfunktionen durchführt.

Haskell bietet für diese Art von verketteten Aktionen "syntaktischen Zucker" in Form der `do`-Notation an:

```
do x <- getLine
   putStr x
```

Den Pfeil kann man sich als Zuweisung vorstellen, wobei tatsächlich nur eine anonyme Funktion erzeugt wird, die einen Parameter `x` an die nächste Anweisung weiterreicht.

3.2 Simulation iterativer Konstrukte

Ein häufiger Anwendungsfall ist das wiederholte Lesen von Eingabe, Manipulation, und Ausgabe. Dabei soll dieser Vorgang wiederholt werden, bis z. B. `quit` eingegeben wird. Dies muss man stets rekursiv machen, da eine Iteration in funktionalen Sprachen nicht (bzw. "nicht direkt") möglich ist. Eine allgemeines "Konstrukt" wäre jedoch wünschenswert, so dass man nicht stets die Rekursion neu implementieren muss:

```
repeating :: (String -> Maybe String) -> IO (Maybe String)
repeating f =
  do x <- getLine
     continue (f x)
     where continue (Just val) =
             do putStr val
                repeating f
           continue (Nothing) =
             do return Nothing
```

Die Funktion `repeating` erhält als Argument eine Funktion, die eine Zeichenkette erwartet und eventuell eine Zeichenkette liefert. Es wird zunächst ein Wert gelesen und anschließend per Musterabgleich entschieden werden, ob die Abarbeitung fortgeführt werden soll.

Beispiel:

```
repeating (\x -> if x == "quit"
                 then Nothing
```

```
                    else Just (x ++ "\n"))
```

Hierbei wird geprüft, ob die Eingabe identisch zu `"quit"` ist; falls ja, wird die Ausführung abgebrochen. Andernfalls wird `x`, gefolgt von einem Zeilenumbruch, zurückgegeben.

Eine allgemeine "Schleife" ist dadurch aber nicht entstanden. In gängigen iterativen Programmiersprachen wird in `while`-Schleifen zunächst eine beliebige Bedingung geprüft, anschließend eine beliebige Folge von Aktionen ausgeführt. Dies lässt sich auch ähnlich in Haskell umsetzen:

```
whileLoop :: Monad m => m Bool -> m a -> m ()
whileLoop cond body =
  do res <- cond
     if res then do body
                    whileLoop cond body
            else return ()
```

Die "Zuweisung" von `cond` an `res` ist notwendig, da sonst `cond` nicht ausgeführt würde.

Beispiel:

```
whileLoop (do res <- isEOF
              return (not res))
          (do line <- getLine
              putStrLn line)
```

Dieser Code kopiert (eine Funktion `isEOF` vorausgesetzt) die Eingabe in die Ausgabe, bis das Ende der Eingabe erreicht ist.

Wie `repeating` ist auch `whileLoop` keine allgemeine Schleife. Das Problem bei `whileLoop` ist, dass man Werte von einer Ausführung des Rumpfs nicht an die zweite Ausführung weiterreichen kann. Außerdem könnte auch die Ausführung der Bedingung ein Resultat haben, dass nicht an den Rumpf übergeben werden kann; die Entwicklung einer solchen Funktion übersteigt aber den Rahmen dieser Arbeit.

4 Fazit

Haskell ist eine sehr mächtige funktionale Sprache, die sich universell einsetzen lässt. Dank des Monadenkonzepts ist Ein- und Ausgabe sehr einfach zu handhaben. Für Typklassen bestehen allerdings noch einige offene Probleme, die aufgrund des Alters des aktuellen Standards noch nicht gelöst sind. Diese und einige andere Themen werden aber im nächsten Standard, "Haskell Prime", aufgegriffen.

Aufgrund der vielfältigen Erweiterungen zum grundlegenden Sprachumfang und der umfassenden Standardbibliothek *Prelude* wird Haskell durchaus als "real world"-Programmiersprache verwendet. Typklassen ermöglichen objektorientierte, Monaden imperative Programmierstile. So gibt es, um nur ein Beispiel zu nennen, einen Fenstermanager für den X-Server, der in Haskell geschrieben ist.

Quellenverzeichnis

[1] Paul Hudak: *The Haskell School of Expression – Learning functional programming through multimedia,* Cambridge University Press 2000

[2] Simon Thompson: *Haskell – The Craft of Functional Programming,* Second Edition, Addison Wesley 1999

[3] Theodore Norvell: *Monads for the Working Haskell Programmer – a short tutorial*, abrufbar unter `http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm`

[4] *The Haskell 98 Report*, abrufbar unter `http://www.haskell.org/onlinereport/`