Proseminar: Perlen der Informatik II
Haskell
List comprehensions & Typklassen

Dmitriy Traytel

8. Juli 2008

Diese Ausarbeitung im Rahmen des Proseminars "Perlen der Informatik II" beschäftigt sich mit einigen Besonderheiten der funktionalen Programmiersprache Haskell. Konkret werden im ersten Teil die sogenannten "list comprehensions" besprochen. Im zweiten Teil werden Typklassen und sinnvolle Erweiterungen dieser betrachtet. Hierfür wird ein Grundverständnis der Prinzipien der funktionalen Programmierung vorausgesetzt.

1 List comprehensions

Komprehension bedeutet laut Duden "Zusammenfassung, Vereinigung von Mannigfaltigem zu einer Einheit". In der Mathematik wird dieser Begriff für die Mengenbildung durch Zusammenfassen von Elementen mit gleichen Eigenschaften verwendet. Von dem englischen Äquivalent "comprehension" leitet sich der Begriff der "list comprehensions" ab. Dieser wird in der deutschen Literatur oft als "Listenbeschreibungen" übersetzt, was nach meiner persönlichen Meinung nicht sehr passend ist. Im Folgenden wird der englische Begriff bzw. die Abkürzung "LC" verwendet.

1.1 Syntax

Die LCs ermöglichen die Konstruktion von Listen, die auf bestehenden Listen basieren. Die Syntax ist dabei an der mathematischen Notation für Mengendefinitionen orientiert.

1.1.1 Einführendes Beispiel

Die Syntax der LC lässt sich am besten zunächst an einem Beispiel erläutern. Wir betrachten dafür die mathematische Schreibweise der Menge aller Teiler der Zahl $n \in \mathbb{N}$:

$$\{x \mid x \in [1, \ldots n],\ n\ mod\ x = 0\}$$

In Haskell kann man dies dank LC auf sehr ähnliche Weise ausdrücken:

```
1  divisors :: Int -> [Int]
2  divisors n = [x | x <- [1 .. n], n `mod` x == 0]
```

$[1..n]$ ist dabei eine verkürzte Notation für eine Liste der natürlichen Zahlen von 1 bis n. Interpretieren kann man die Funktion *divisors* als die Anweisung "man nehme die natürlichen Zahlen von 1 bis n und schreibe nur diejenigen in eine Liste, welche n ohne Rest teilen".

1.1.2 Formale Syntax

Formal sieht in Haskell jede LC folgendermaßen aus[1]:

$$lcExp \rightarrow [exp \mid qual_1, \ldots, qual_n] \text{ für } n \geq 1$$

Dabei ist exp ein **Ausdruck** und die $qual_i$ sind sogenannte **Qualifikatoren**[2], die selber

- **Generatoren** der Form $p \leftarrow listExp$ mit einem Muster p und einem Ausdruck $listExp :: [a]$ vom Typ Liste
- **Tests** der Form $boolExp$ mit dem boolschen Ausdruck $boolExp$

sein können.

1.1.3 Weiterführende Beispiele

Mit Hilfe von LC kann man viele Probleme auf eine sehr übersichtliche und dem mathematischen Verständnis entsprechende Weise darstellen. Interessant ist z.B. die Primzahlfindung mit dem Sieb des Eratosthenes.

```
1  primes :: [Int]
2  primes = sieve [2 .. ]
3           where
4           sieve :: [Int] -> [Int]
5           sieve [] = []
6           sieve (x:xs) = x : sieve [y | y <- xs, y `mod` x /= 0]
```

Dabei wird eine unendliche Liste von Primzahlen erstellt, die später mit den eingebauten Listenfunktionen (z.B. *take* 5 *primes* liefert die ersten 5 Primzahlen und *primes*!!4 die 4. Primzahl der Liste) verarbeiten werden kann. Dies ist überhaupt nur möglich, weil Haskell alle Funktionen "lazy" auswertet. D.h. die Funktion *primes* wird im ersten Beispiel nur bis zum 5. Listenelement (im zweiten sogar nur bis zum 4.) berechnet.
Das bekannteste Haskell-Beispiel ist aber wohl der dreizeilige Quicksort-Algorithmus hier zunächst in einer nur auf Integer-Listen eingeschränkter Form (die allgemeinere Form kommt etwas später in der Ausarbeitung):

```
1  qsort:: [Int] -> [Int]
2  qsort [] = []
3  qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++ qsort [y | y <- xs, y > x]
```

[1] [2] p.270
[2] [6] p.344

Alle bisher genannten Beispiele arbeiten mit jeweils einem Generator und einem Test. Natürlich ist aber auch das öftere Vorkommen zugelassen. Nehmen wir eine Funktion, die Primzahlen, welche sich als Summe zweier Quadratzahlen darstellen lassen und die Wurzeln aus ebendiesen Quadratzahlen ausgibt:

```
1  primesAsSqSum :: Int -> [(Int,Int,Int)]
2  primesAsSqSum n =
3          [(p,x1,x2) | p <- ps, x1 <- xs, x2 <- xs, x1 <= x2, x1*x1 + x2*x2 == p]
4                  where
5                  ps = take n primes
6                  xs = [1 .. n]
```

Hier werden gleich drei Generatoren und zwei Tests benutzt. Zudem werden die generierten Werte, die beide Tests bestehen, zu einem Tripel zusammengefasst. Dies entspricht ebenfalls der LC-Syntax, da links von dem "|" beliebige Ausdrücke stehen können (sofern sie mit den angegebenen Typen übereinstimmen).

1.2 Semantik

Um mit LC-Ausdrücken richtig arbeiten zu können, muss man verstehen, wie diese vom Haskell-Compiler ausgewertet werden.

1.2.1 Auswertung von LC-Ausdrücken

Der Compiler benutzt für die Auswertung drei Transformationsregeln[3]:

1. $[exp \mid qual_1,\ qual_{2\ldots n}] \mapsto concat\ [[e \mid qual_{2\ldots n}] \mid qual_1]$

   Mit dieser Transformation wird ein Qualifikator eliminiert. Man sollte beachten, dass die Auswertung von links nach rechts abläuft, d.h. $qual_1$ wird als erstes evaluiert. *concat* ist dabei eine Haskell-Funktion, die die Elemente einer Liste von Listen aneinanderhängt. Durch mehrmaliges Ausführen dieser Transformation wird erreicht, dass rechts von "|" nur ein einzelner Qualifikator steht. Dann wird eine der folgenden Transformationen angewendet.

2. $$[exp \mid p \leftarrow listExp] \mapsto \begin{array}{ll} \textbf{let} & ok\ p\ =\ True \\ & ok\ \_\ =\ False \\ \textbf{in} & map\ (\backslash p \rightarrow exp)\ (filter\ ok\ listExp) \end{array}$$

   Diese Transformation eliminiert einen einzelnen Generator. Die $filter$-Funktion überprüft auf welche Elemente der Liste das Muster p passt. Auf diese wird dann jeweils durch *map* die Funktion $(\lambda\ p.\ exp)$ angewendet (in Haskell entspricht "$\backslash \ldots$", dem Lambda-Kalkül Term $\lambda \ldots$). In den obigen Beispielen ist die $filter$-Funktion redundant, da das Muster x auf jedes Listenelement passt. Interessanter wird es bei anderen Mustern wie z.B. $[1 \mid False \leftarrow [1/=1,\ 2+2==5,\ 2>1,\ 2+2==4]]$. Hier passieren nur die Ausdrücke den Filter, die den boolschen Wert $False$ haben. Nach der Anwendung von $(\lambda\ False.\ 1)$ ist das Ergebnis der Berechnung $[1, 1]$

[3] [2] p.190

3. $[exp \mid boolExp] \mapsto$ **if** $boolExp$ **then** $[exp]$ **else** []

   Diese Transformation eliminiert einen einzelnen Test auf eine selbsterklärende Weise. Es bleibt nur noch zu sagen, dass die **if**...**then**...**else**... Anweisung nicht Teil des Haskell-Kernsprache ist sondern intern durch eine **case**...**of**... Anweisung ersetzt wird, was für diese Betrachtung aber irrelevant ist.

Es folgt ein Beispiel für die Transformation eines LC-Ausdrucks:
Man betrachtet eine Funktion, der die ersten 20 Quadratzahlen berechnet:

```
squares = [x*x | x<-[1 .. 50], x<=20]
```

Nun wird die **Transformation 1** ausgeführt:

```
squaresStep1 = concat[ [x*x | x<=20] | x<-[1 .. 50] ]
```

Danach wird der äußerste Qualifikator mit **Transformation 2** eliminiert:

```
squaresStep2 =
        concat (
        let     ok x = True
                ok _ = False
        in map (\x-> [x*x | x<=20]) (filter ok [1 .. 50])
        )
```

Schließlich wird der die **Transformation 3** auf den übriggebliebenen LC-Ausdruck angewendet:

```
squaresStep3 =
        concat (
        let     ok x = True
                ok _ = False
        in map (\x-> if x<=20 then [x*x] else []) (filter ok [1 .. 50])
        )
```

Damit ist der LC-Ausdruck vollständig aufgelöst.

1.2.2 Gebrauch von LC-Ausdrücken

LC-Ausdrücke sind syntaktischer Zucker für die funktionale Programmierung, da sie eine oftmals kompaktere Schreibweise bieten. Außerdem erlauben sie einem Menschen ohne Kenntnisse der funktionalen Programmierung, wohl aber mit mathematischen Vorwissen einen leichteren Einstieg in Haskell. So können z.B. *map* und $filter$-Funktionen als Funktionen höherer Ordnung, die beim ersten Mal schwierig zu begreifen sind, ganz durch LC-Ausdrücke ersetzt werden ($map\ f\ xs$ entspricht $[f\ x \mid x \leftarrow xs]$ und $filter\ b\ xs$ entpricht $[x \mid x \leftarrow xs, b\ x]$).
Man kann aber eben auch ganz auf LC-Ausdrücke verzichten, denn sie lassen sich durch die oben genannten Transformationen einfach eliminieren. Oft kann man die Transformationen so optimieren und verkürzen, dass auch ohne LC eine ansprechende Form der Programms sich erreichen lässt. Es folgen die ersten drei Beispielfunktionen in denen LC-Ausdrücke durch *map* und $filter$ ersetzt wurden.

```haskell
divisorsNoLC :: Int -> [Int]
divisorsNoLC n = filter ((==0) . (n `mod`)) [1 .. n]

primesNoLC :: [Int]
primesNoLC = sieve [2 ..]
        where
    sieve :: [Int] -> [Int]
        sieve [] = []
    sieve (x:xs) = x : sieve (filter ((/= 0) . (`mod` x)) xs)

quicksortIntNoLC :: [Int] -> [Int]
quicksortIntNoLC [] = []
quicksortIntNoLC (x:xs) = quicksortIntNoLC (filter (<=x) xs)
                        ++ [x] ++ quicksortIntNoLC (filter (>x) xs)
```

Man merkt bereits bei diesen einfachen Beispielen, dass die Ausdrücke ohne tiefere Kenntnisse von Haskell kaum zu verstehen sind (wie z.B. $filter\ ((/=0)\ .\ (`mod`\ x))\ xs$). Die Komplexität des LC-freien Programm nimmt dabei mit der Komplexität und der Anzahl der Qualifikatoren zu. So lässt sich das vierte Beispiel natürlich auch in eine LC-freie Form bringen, dies sei aber hier dem interessierten Leser überlassen.

2 Typklassen

Bei dem Beispiel *qsort* haben wir den Typ der Listen auf Integer eingeschränkt. Also bedarf es einer weiteren Implementierung, wenn man auch Buchstaben alphabetisch sortieren will. Der einzige Unterschied zu der obigen Implementierung ist in der Typannotation zu finden:

```
1  qsortChar :: [Char] -> [Char]
2  qsortChar [] = []
3  qsortChar (x:xs) = qsortChar [y | y <- xs, y <= x]
4                           ++ [x]  ++ qsortChar [y | y <- xs, y > x]
```

Zunächst können wir die Funktion nicht auch *qsort* nennen, worauf wir noch später zu sprechen kommen. Außerdem haben wir den gleichen Algorithmus zweimal im Quellcode, was aber vor allem für Wartungszwecke sehr inpraktikabel ist. Deswegen sollte man nicht nur aus Gründen der Faulheit auf ungewollte Redundanz im Code verzichten. Man könnte den Versuch unternehmen die Typannotation zu $qsort :: [a] \rightarrow [a]$ zu generalisieren, dies scheitert aber daran, dass nicht auf allen Typen die Ordnungsrelation $\leq$ besteht. Für die Lösung des Problems ist in Haskell das Konzept der Typklassen vorgesehen.

2.1 Hindley-Milner Typsystem

Haskell verwendet das Hindley-Milner Typsystem, was den Vorteil mit sich bringt, ganz auf Typannotationen verzichten zu können. Nach dem Typ-Inferenz Algorithmus von Hindley und Milner (auf diesen wird hier nicht genauer eingegangen, es soll nur gesagt werden, dass er auf dem bekannten Unifikationsalgorithmus aufbaut) werden die allgemeinsten Typen automatisch vom System bei der Übersetzung korrekt erkannt. Darin liegt aber auch der Nachteil dieses Typsystems: die Typen müssen zur Übersetzungszeit eindeutig bestimmbar sein und damit sind dynamische Bindungen nicht möglich.

2.2 Polymorphismus

Der Begriff der dynamischen Bindungen ist eng verknüpft mit dem Begriff des Typenpolymorphismus. Es gibt mehrere Arten des Polymorphismus. Für diese Betrachtung sind jedoch nur der parametrische und der ad-hoc Polymorphismus relevant.

2.2.1 parametrischer Polymorphismus

Die parametrische Polymorphie erlaubt Typvariablen in den Funktionsdefinitionen (in Java entspricht dies den Generics). Haskell (und alle anderen funktionalen Programmiersprachen die das Hindley-Milner Typsystem verwenden, wie z.B. ML, Miranda) sowie auch der typisierte λ-Kalkül benutzen dieses Polymorphiekonzept. Beispiele für parametrisch polymorphe Funktionen in Haskell sind unteranderem die *length* und die *map* - Funktionen:

```haskell
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Der parametrische Polymorphismus gibt dem Programmierer die Möglichkeit eine einzige Funktionsdefinition auf unterschiedliche Typen anzuwenden.

2.2.2 ad-hoc Polymorphismus

Die ad-hoc Polymorphie wird auch "Überladen" genannt. Sie ermöglicht Funktionen mit gleichen Namen aber unterschiedlichen Definitionen. Der Compiler kann anhand der Typen die anzuwendende Funktion auswählen (so funktioniert es z.B. in Java). In Haskell würde das so aussehen:

```haskell
(==) :: Int -> Int -> Bool
0 == 0 = True
a == 0 = False
a == b = a - b == 0

(==) :: Bool -> Bool -> Bool
a == b = (a && b) || ((not a) && (not b))
```

Dies ist aber **falsch** in Haskell. Die einzelnen Funktionen für sich stimmen zwar, aber eine solche Namensüberladung ist verboten. Man müsste die Namen etwa zu $== Int$ und $== Bool$ ändern, was aber sehr unhandlich ist. Abhilfe gegen diese Einschränkung schafft eine Erweiterung des Hindley-Milner Typsystems - die Typklassen.

2.3 Syntax

Die Idee der Typklassen ist die Folgende: "gruppiere die Typen auf die eine bestimmte Funktion (die typabhängig implementiert werden soll) angewendet werden soll in einer Klasse".

2.3.1 Klassen und Instanzen

Für das letzte Beispiel ist es sinvoll die "Gleichheitsklasse" **Eq** (für *equality*) für alle Typen die auf Gleichheit getesten werden sollen, zu erstellen:

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

Die erste Zeile beinhaltet hier den Namen der Klasse, die zweite die sogenannte **Signatur**, die Funktionen enthält, welche von den Klassenmitgliedern zwangsweise implementiert werden müssen (es sei denn, es sind bereits Definitionen vorhanden). Jeder Typ, der der Klasse Eq "beitreten" will (bzw. eine **Instanz** der Klasse werden will) muss also die Funktion (==) implementieren - dies erinnert stark an die Interfaces in Java. Mit dem obigen Beispiel ergibt sich:

```
instance Eq Int where
        0 == 0 = True
        a == 0 = False
        a == b = a - b == 0

instance Eq Bool where
        a == b = (a && b) || ((not a) && (not b))
```

Hiermit sind Int und Bool Instanzen der Klasse Eq.

2.3.2 Abgeleitete Klassen und Funktionen

Darüber hinaus kann man Typklassen aus bereits bestehenden Klassen ableiten. Die Syntax dafür ist am Beispiel der Klasse für "geordnete" Typen (d.h. mit existierender Ordnungsrelation) sichtbar:

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

Der dem Operator $\Rightarrow$ vorhergehende Typ, von dem der neue Typ abgeleitet wird, heißt **Kontext**. Die Notation wird interpretiert als: ein Typ wird genau dann zur Instanz der Klasse Ord, wenn er Instanz der Klasse Eq ist und die Funktion $(<) :: a \rightarrow a \rightarrow Bool$ implementiert. Außerdem können die Funktionen $<=$, $>$ und $>=$ überschrieben werden, aber sie müssen es nicht, da bereits eine logisch richtige Implementierung vorhanden ist. Nun kann man Funktionen für ganze Klassen von Typen definieren: es folgt die letzte Betrachtung von Quicksort:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (pivot:xs) = quicksort [x | x<-xs, x<=pivot]
                         ++ [pivot] ++ quicksort [x | x<-xs, x>pivot]
```

Dieser Kontext fordert, dass der Typ a eine Instanz der Klasse Ord ist.

2.3.3 Multiple constraints

Bis jetzt wurde immer jeweils nur eine Typforderung an eine abgeleitete Klasse, Instanz einer Klasse oder Funktion betrachtet. Im Kontext sind aber auch mehrere Forderungen möglich (sogenannte **multiple constraints**[4]):

```
instance (Eq a, Eq b) => Eq (a,b) where
        (m,n) == (x,y) = m==x && n==y
```

Das Beispiel für eine abgeleitete Klasse mit multiple constraints ist die "Vaterklasse" für numerische Typen (außer dem eingebauten Typ Int) in Haskell (die Klasse Show wird nachher noch kurz erläutert):

[4] [6] p.218

```
class (Eq a, Show a) => Num a where
        (+), (-), (*) :: a -> a -> a
        negate :: a -> a
        abs, signum :: a -> a
        fromInteger :: Integer -> a
        x - y = x + (negate y)
```

2.3.4 Eingebaute Klassen

In der Haskell-Prelude sind viele Typklassen bereits eingebaut. Dazu gehören die bereits in einer verkürzten Version vorgestellte **Eq** und **Ord** (in der Prelude-Version gibt es in Eq noch die definierte Funktion $(/=)$ und in Ord die Funktionen *min*, *max* und *compare*). Außerdem gibt es noch eine ganze Reihe von numerischen Klassen (**Real**,**Integral**, **Fractional** usw.), die alle von der Klasse **Num** abgeleitet sind, die Klasse **Enum** für Aufzählungstypen und die Klassen **Show** (stellt die *show* Funktion zur Verfügung, die der Java-Methode *toString*() entspricht) und **Read** die umgekehrt zu show den Ausdruck des gesuchten Typs aus einem String mit der Funktion *read* parst[5].

Es bleibt noch zu sagen, dass die Typklassen nur eine geringe Änderung des Hindley-Milner Typinferenz-Algorithmus hervorrufen, sodass weiterhin auf Typannotationen verzichtet werden kann. Die dynamische Bindung bleibt damit aber auch nicht erlaubt (d.h. konkret kann es z.B. keine Listen vom Typ $Eq\ a \Rightarrow [a]$ mit unterschiedlichen Typen innerhalb der Liste, wie $[42,\ True,\ 'c']$ geben).

2.4 Erweiterte Typklassen

So schön die Typklassen auch sind - sie können nicht alle wünschenswerten Typenfamilien abdecken. Es ist aber durchaus möglich durch bereits geringe Erweiterungen des Konzepts eine größere Abdeckung zu erzielen.

2.4.1 Konstruktor-Typklassen

Das Typsystem von Haskell nimmt die folgende Einteilung der Typen nach **Arten**[6] vor:

- Von der Art $*$ sind die **Grundtypen** wie *Int* oder *Bool*
- Von der Art $\kappa_1 \rightarrow \kappa_2$ sind **Konstruktoren** (auch genannt Typen höherer Ordnung[7]) wobei κ_1 und κ_2 wieder Arten sind (Beispiele dafür sind Konstruktoren abstrakter Datentypen wie z.B. Listen (sie haben die Art $* \rightarrow *$) oder Funktionen $a \rightarrow b$ (deren Art ist $* \rightarrow * \rightarrow *$))

[5] diese Liste hat keinen Anspruch auf Vollständigkeit sondern soll nur die wichtigsten Klassen nennen, eine vollständige Übersicht gibt es unter [1] `http://www.haskell.org/onlinereport/basic.html#sect6.3`

[6] [4] p.264

[7] [4] p.249

Die bisherige Definition der Typklassen erlaubte lediglich das Zusammenfassen von Grundtypen zu Klassen. Die sogennannten Konstruktor-Typklassen erlauben auch eine Gruppierung der Konstruktor-Typen.

Die Functor-Klasse

Man betrachte zunächst folgende abstrakte Datentypen:

```
data Tree a =
        Leaf a | Node a (Tree a) (Tree a)
        deriving (Show,Eq,Ord,Read)

data Maybe x = Just x | Nothing
        deriving (Show,Eq,Ord,Read)
```

Das Schlüsselwort *deriving* erlaubt es, die abstrakten Datentypen zu Instanzen der angegebenen eingebauten Typklassen zu machen (nur von eingebauten Klassen kann automatisch abgeleitet werden). Die map-Funktionen für diese Datentypen werden folgendermaßen definiert:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node x l r) = Node (f x) (mapTree f l) (mapTree f r)

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f (Just x) = Just (f x)
mapMaybe f Nothing = Nothing
```

Nun sieht man, dass sich die Typen von $mapTree$, $mapMaybe$ und auch vom einfachen *map* über Listen (vom Typ $(a \to b) \to [a] \to [b]$) sehr ähneln. Will man die Datentypen, auf denen die *map*-Funktionen operieren können, zu einer Klasse zusammenfassen, so wird man mit der bisherigen Definition der Klassen an das Problem der Wahl des gemeinsamen Grundtyps stoßen. Wenn man aber die Erfassung von Typen der Art $* \to *$ in Klassen zulässt, kann man den gemeinsamen Konstruktor-Typ t bei den *map*-Funktionen definieren, sodass der allgemeine Typ von diesen Funktionen $(a \to b) \to t\ a \to t\ b$ ist. Da die Konstruktor-Typen bereits Bestandteil von Haskell98 (die standardisierte Haskell Version) sind, wird genau so wie eben beschrieben die Functor-Klasse definiert. In ihr werden die Datentypen, auf welchen die *map*-Funktion definiert werden soll bzw. ist, zusammengefasst. Die Syntax entspricht dabei der Typklassen-Syntax:

```
class Functor f where
        fmap :: (a -> b) -> f a -> f b
```

Nun kann man auf bekannte Weise Instanzen zu der Klasse-Funktor hinzufügen:

```
instance Functor [] where
        fmap = map

instance Functor Tree where
        fmap f (Leaf x) = Leaf (f x)
        fmap f (Node x t1 t2) = Node (f x) (fmap f t1) (fmap f t2)
--oder einfach: fmap = mapTree

instance Functor Maybe where
                fmap f (Just x) = Just (f x)
```

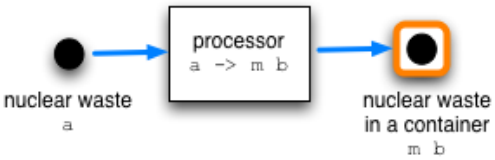
```
11                  fmap f Nothing = Nothing
12  --oder einfach: fmap = mapMaybe
```

Hiermit erhält man eine Klasse, deren Instanzen Datentypen sind, die eine *map*-Funktion definieren.

Die Monad-Klasse

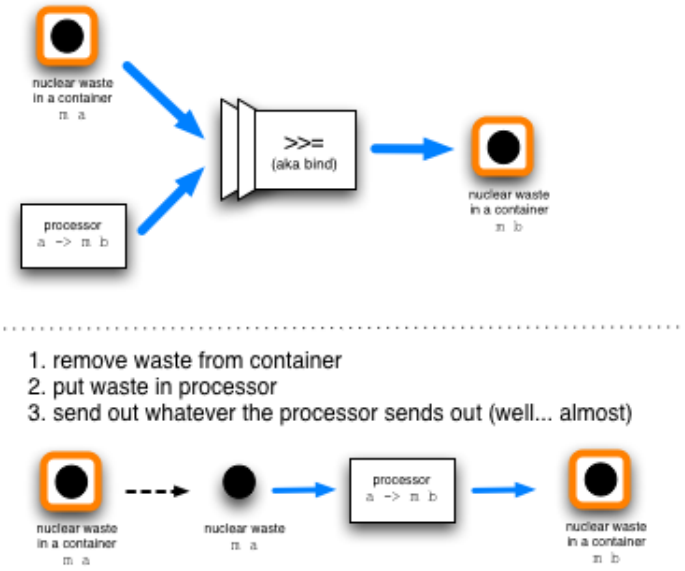
Monaden sind ein weiteres Beispiel für die Konstruktor-Klassen. Der Begriff der Monaden kommt aus der Kategorientheorie, dieser Ursprung ist aber für eine erste Betrachtung im Kontext der funktionalen Programmierung irrelevant. Ein gutes Beispiel, um ein Gefühl für die Monaden zu bekommen, ist die **Atommüll-Metapher**[8]:
Man stelle sich eine **Fabrik** vor die **Atommüll** in unterschiedlichen Etappen verarbeitet. Ein **Prozessor** nimmt Atommüll und spuckt verarbeiteten Atommüll aus.

Moderne Prozessoren können den Atommüll zusätzlich in ein Container packen.



Der Atommüll entspricht den Daten, die Prozessoren Funktionen und moderne Funktionen sind Funktionen deren Rückgabetyp ein Typ höherer Ordnung ist. Die gesamte Fabrik ist eine Monade. Daneben gibt es sogenannte **bind-Roboter**, die verpackten Atommüll in einen modernen Prozessor stecken und dann die Rückgabe des Prozessors weiterleiten.

[8] [5] Folien 5-9



Diese Roboter stehen für die zentrale Funktion der Monad-Klasse: die bind Funktion. Durch das Hintereinanderschalten der bind-Roboter bekommen wir die gesamte **Sequenz** von Stationen der Atommüllverarbeitung.
Warum braucht man nun dieses umständliche Modell? Für bestimmte Programmabläufe braucht man eine streng festgelegte Reihenfolge der Berechnung. Wie zum Beispiel soll der Ausdruck $inputInt - inputInt$[9] ausgewertet werden? Der Wert dieses Ausdrucks hängt nämlich davon ab, ob die zuerst eingegebene Zahl der Minuend oder der Subtrahend ist (*inputInt* liest ein Integer vom der Standardeingabe). In den nicht-funktionalen Programmiersprachen ist diese gewünschte Sequenzierung implizit vorhanden (dies bewirken Seiteneffekte). Haskell dagegen ist eine reine funktionale Programmiersprache, in der es einfach keine Seiteneffekte gibt. Die gewünschte Reihenfolge muss man also explizit angeben. Dies können die Monaden bewerkstelligen. Für das Beispiel mit der Eingabe existiert in Haskell das do-Konstrukt, das selber ebenfalls auf Monaden beruht:

```
1  do
2          e <- inputInt
3          f <- inputInt
4          return (e-f)
```

Die eigentliche Monad-Klasse ist folgendermaßen definiert:

```
1  class Monad m where
2          (>>=) :: m a -> (a -> m b) -> m b        --bind
3          (>>) :: m a -> m b -> m b                --sequence
4          return :: a -> m a
5          fail :: String -> m a
6
7          m >> k = m >>= \_ -> k
8          fail s = error s
```

[9] [6] p.402

Beispiele für Instanzen der Monad-Klasse sind Listen und die Option Maybe:

```
1  instance Monad Maybe where
2          Just x >>= k = k x
3          Nothing >>= k = Nothing
4          return = Just
5          fail s = Nothing
6
7  instance Monad [] where
8          xs >>= f = concat (map f xs)
9          return x = [x]
10         fail x = []
```

Wie bereits angedeutet ist das gesamte Haskell-IO durch Monaden implementiert (inklusive der do-Notation). Dieses Beispiel würde aber für eine erste Behandlung der Monaden zu tief gehen.

2.4.2 Ausblick: Typklassen mit mehreren Parametern

Eine weitere denkbare Erweiterung der Typklassen ist die Zulassung mehrerer Parameter. Diese Erweiterung ist aber nicht Teil von Haskell98, deswegen wird hier nur die Grundidee angesprochen.
Sinnvoll sind die Typklassen mit mehreren Parametern, wenn man z.B. die Vektorräume als Typ darstellen will[10]. Hier benötigt man neben der abelschen Gruppe, die die Vektoren stellt, noch einen Körper mit Skalaren. In Haskell würde es ungefähr so aussehen:

```
1  class (Fractional scalar) => VectorSpace vektor scalar where
2          ...
```

Hier ist man aber noch nicht fertig. Bei Typklassen mit mehreren Parametern muss man nämlich auf funktionale Abhängigkeiten der Parameter aufpassen. Es kann nämlich passieren, dass mit der Wahl des ersten Parameters, der zweite bereits festgelegt ist. Dies ist auch bei den Vektorräumen der Fall: der Vektor legt automatisch den ihm zugrunde liegenden Körper fest. Mit der Annotation der funktionalen Abhängigkeiten sieht die Vektorraumklasse folgendermaßen aus:

```
1  class (Fractional scalar) => VectorSpace vektor scalar | vektor -> scalar where
2          ...
```

Es sind auch weitere sinvolle Anwendungsbeispiele denkbar, wie z.B. die indizierten Listen oder ähnliche nummerierte Datentypen, diese würden aber auch hier zu weit gehen.

[10] [3] Folie 32-34

Literatur

[1] Haskell 98 language and libraries the revised report. Technical report, February 1999. `http://www.haskell.org/onlinereport/`.

[2] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

[3] Sven M. Hallberg. *Vortrag "Haskell - A Wild Ride" im Rahmen des Bildungswerkes des CCC Hamburg*. 2007. `https://wiki.hamburg.ccc.de/images/8/86/Pesco04haskell_slides_de.pdf`.

[4] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[5] Marc Rehmsmeier. *Vorlesung "Programmieren in Haskell WS 2006/2007" an der Universität Bielefeld - Folien zu Kapitel 10: Monaden*. 2007. `www.techfak.uni-bielefeld.de/ags/pi/lehre/AuDIWS06/folien_hs/Kapitel10_Monaden.pdf`.

[6] Simon Thompson. *Haskell The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.