Higher-Order Funktionen und Auswertungsstrategie in Haskell

Proseminar Fortgeschrittene Programmierkonzepte

Martin Ziegler, Erik Wrede

28. Juni 2019

1 Einleitung

Mit Haskell als funktionale Programmiersprache können schnell und effizient viele Probleme gelöst werden. Dabei sind bestimmte Werkzeuge und Strukturen hilfreich, um schnell und übersichtlich mächtige, aber verständliche Programme zu erstellen. Diese Arbeit soll eine Einführung in den Umgang mit Funktionen höherer Ordnung sein und ein tiefergehendes Wissen über die Auswertungsstrategie in Haskell vermitteln. Dieses Wissen ist grundlegend für die Entwicklung kompakter, mächtiger und effizienter Haskell-Programme und erspart eigenen Implementierungsaufwand. Um das hier übermittelte Wissen sinnvoll anwenden zu können, empfiehlt es sich, die in der Vorlesung Programmierung vorgestellten Grundlagen von Haskell zu kennen.

2 Higher-Order Funktionen

Bei der Arbeit mit Daten in Haskell gibt es verschiedene Schritte zur Auswertung und Verarbeitung dieser Daten. Viele basieren auf der rekursiven Traversierung einer Datenstruktur, um zum Beispiel Daten zu filtern, transformieren oder zusammenzufassen.

Listing 1: Filtern einer Liste von Altersdaten nach Volljährigkeit und Rentenalter

```
1 filterOfAge :: [Int] -> [Int]
2 filterOfAge [] = []
3 filterOfAge (x:xs) | x >= 18 = x : filterOfAge xs
4                    | otherwise = filterOfAge xs
5
6 filterNotRetired :: [Int] -> [Int]
7 filterNotRetired [] = []
8 filterNotRetired (x:xs) | x < 67 = x : filterNotRetired xs
9                         | otherwise = filterNotRetired xs
```

Diese beiden Funktionen sind ähnlich aufgebaut: Sie traversieren eine Liste von Zahlen rekursiv und filtern diese nach bestimmten Kriterien. Bis auf die Wahl der Filterkriterien unterscheiden sie sich allerdings kaum.

Um die Verwendung dieser ähnlichen Funktionen effizienter zu gestalten, gibt es in Haskell Funktionen höherer Ordnung, sogenannte *Higher-Order Functions*. Im Wesentlichen gibt es zwei Typen von Funktionen höherer Ordnung: Zum einen sind das Funktionen, die eine andere Funktion als Parameter übergeben bekommen und Funktionen, die eine Funktion als Rückgabewert definieren, beispielsweise durch *Currying*. Außerdem gibt es Higher-Order Funktionen, die beide

dieser Eigenschaften erfüllen.
Die hier vorgestellten Higher-Order Funktionen sind alle in `Prelude` vordefiniert.

2.1 Map und Filter

Zwei häufig verwendete Higher-Order Funktionen sind `map` und `filter`.

filter Die Funktion `filter` wendet eine Filterfunktion vom Typ `a -> Bool` rekursiv auf alle Elemente einer übergebenen Liste an. Gibt die übergebene Filterfunktion für ein Element der Liste `True` zurück, so verbleibt das Element in der zurückgegebenen Liste - bei `False` wird es entfernt. Die Funktion ist wie folgt definiert:

Listing 2: Definition filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) | f x = x : filter f xs
                | otherwise = filter f xs
```

Mit der Higher-Order Funktion `filter` können die Beispiele aus Listing 1 mit deutlich geringerem Implementierungsaufwand und ohne redundanten Code realisiert werden:

Listing 3: Beispiel zu filter

```
filterOfAge = filter (\x -> x >= 18)
filterNotRetired = filter (\x -> x < 67)
```

Listing 4:

```
filterOfAge [2,5,18,24,4,56]
= [18,24,56]
```

map Die Funktion `map` wendet eine übergebene Transformationsfunktion auf alle Elemente der Liste an. So können einfach Transformationen auf viele Elemente angewendet werden [Gie16].

Listing 5: Definition map

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

Wenn `map` und `filter` kombiniert werden, können schnell Transformationen auf viele Werte angewendet werden und unerwünschte Werte herausgefiltert werden. So können die im vorigen Beispiel gefilterten Altersdaten durch Abzug des aktuellen Jahres nun auf ihr Geburtsjahr abgebildet werden:

Listing 6:

```
map (\x -> 2019 - x) [18,24,56]
= [2001,1995,1963]
```

2.2 Curry, Uncurry, „.“-Operator

Auch beim Currying können Higher-Order Funktionen hilfreich sein. Beim Currying wird eine Funktion, die mehrere Argumente akzeptiert in eine Reihe von Funktionen umgewandelt, die jeweils nur ein Argument akzeptieren.

Listing 7: Definition curry/uncurry

```haskell
curry :: ((a,b) -> c) -> a -> b -> c
curry f a b = f (a,b)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b
```

Dies ist besonders hilfreich, wenn eine Funktion ein Tupel zurückgibt, welches dann aber von einer gecurryten Funktion weiter verarbeitet wird. Damit diese Funktion dann nicht neu geschrieben werden muss, um Tupel zu akzeptieren, was besonders bei Methoden von Typklassen aufwendiger ist, kann `uncurry` angewendet werden:

Listing 8:

```haskell
partition :: Double -> (Double, Double)
partition x = (x/2, x/2)
Auswertung:
(uncurry (+)) (partition 6.0) = 6.0
```

Verkettungen von Funktionen können in Haskell mit dem Operator „.“ realisiert werden. Dabei muss analog zur mathematischen Funktionskomposition beachtet werden, dass der Rückgabewert der ersten Funktion den selben Typ wie das Argument der zweiten Funktion der Verkettung hat. So können Funktionsaufrufe übersichtlich verkettet werden.

Listing 9:

```haskell
double :: Int -> Int
double a = a + a

double (double 2) = (double . double) 2
```

2.3 fold(r/l) und zipWith

fold Um in Haskell schnell eine Liste auszuwerten, existieren die `fold`-Funktionen. Eine allgemeine `fold`-Funktion traversiert eine Liste und wendet eine Kombinationsfunktion auf alle Elemente der Liste an. Dafür wird der `fold`-Funktion ein Startwert übergeben. Meistens ist dieser Startwert bezogen auf das zu erwartende Ergebnis oder die Kombinationsfunktion neutral gewählt (zum Beispiel 0, 1 oder eine leere Liste). Im nächsten Schritt wird rekursiv jedes Element der Liste durch die Kombinationsfunktion mit dem Startwert kombiniert. Dabei ist der Startwert jedes Rekursionsschrittes das Ergebnis aus der Kombination des vorherigen Startwertes mit dem vorigen Listenelement.

Listing 10: Definition foldr

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Listing 11: Definition foldl

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Ein einfaches Beispiel dafür ist die Berechnung der Listenlänge:

Listing 12:

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

Der Unterschied zwischen `foldr` und `foldl` besteht in der Auswertungsreihenfolge. Mit `foldr` wird die Liste von rechts nach links, mit `foldl` von links nach rechts ausgewertet. Zu beachten ist, dass bei `foldl` und `foldr` die Reihenfolge der Argumente der Kombinationsfunktion vertauscht ist. Dies kann schnell zu Fehlern in Programmen führen. Daher kann eine Kombinationsfunktion nicht immer ohne Modifizierungen sowohl für `foldl` als auch für `foldr` genutzt werden. Im Fall des in Listing 12 genannten Beispiels erfolgt die Zählung der Elemente von rechts nach links. Die Wahl der Auswertungsrichtung führt zum Beispiel bei einer Elementweisen Division zu unterschiedlichen Ergebnissen. Für weitere Informationen sei hier auf die referenzierte Literatur verwiesen.

Auch das Zusammenfügen einzelner Listen zu einer Gesamtliste kann durch eine `fold`-Funktion realisiert werden: Die `concat`-Funktion bekommt eine Liste von Listen mit Elementen des selben Typen übergeben und gibt eine zusammengefügte Liste zurück.
Hier wird als Startwert eine leere Liste gewählt, die Listenverknüpfung (`++`) ist die Kombinationsfunktion. Die Funktion `concat` ist in Haskell vordefiniert und greift auf die bereits vorgestellte `foldl`-Funktion zurück.

Listing 13: Definition und Beispiel concat

```
concat :: [[a]] -> [a]
concat = foldl (++) []

concat [[1,2,3],[4,5,6],[7,8,9]]
= [1,2,3,4,5,6,7,8,9]
```

zipWith Für die elementweise Kombination zweier Listen desselben oder unterschiedlicher Typen mithilfe einer Kombinationsfunktion ist die Funktion `zipWith` in Haskell vordefiniert. Bei der rekursiven Traversierung beider Listen wird das Ergebnis der Kombination der Elemente an einem Index am selben Index einer neuen Liste abgespeichert. Wenn eine Liste länger ist als die andere, wird nur kombiniert, solange noch Elemente in beiden Listen vorhanden sind. In Listing 15 wird so mithilfe von `zipWith` paarweise das arithmetische Mittel zweier Listen von Zahlen berechnet.

Listing 14: Definition zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y) : (zipwith f xs ys)
```

Listing 15: Beispiel zipWith

```
zipWith (\x y -> (x + y)/2) [5,6,7,8,9,9,0,81] [1,2,3,4,5,65,6]
= [3.0,4.0,5.0,6.0,7.0,37.0,3.0]
```

2.4 Higher-Order Funktionen auf anderen Datenstrukturen

Higher-Order Funktionen können in Haskell auch für andere Datenstrukturen als die vordefinierten Listen verwendet werden. Die Methode `fmap`, welche analog zu `map` implementiert werden sollte, ist in der Typklasse `Functor` deklariert. Um diese Methode zu implementieren, muss die Datenstruktur Mitglied in der Typklasse `Functor` sein [Bir15]. Dies wird in Listing 16 exemplarisch anhand eines selbst definierten Binärbaums gezeigt.

Listing 16: Definition BinaryTree

```haskell
data BinaryTree a = Nil | Node a (BinaryTree a) (BinaryTree a)

instance Functor BinaryTree where
    fmap _ Nil = Nil
    fmap f (Node a b c) = (Node (f a) (fmap f b) (fmap f c))
```

Listing 17: Auswertung fmap

```haskell
fmap (\x -> 2019 - x) (Node 18 (Node 24 Nil Nil) (Node 56 Nil Nil))
= Node 2001 (Node 1995 Nil Nil) (Node 1963 Nil Nil)
```

Auch andere Higher-Order Funktionen können auf weitere Datenstrukturen wie zum Beispiel Binärbäume oder Maps angewendet werden. Da bei Funktionen wie `fold` die Art und Reihenfolge der Traversierung einen Einfluss auf das Ergebnis haben kann, und das Löschen von Elementen komplizierter als in einer üblichen Liste ist, kann der Implementierungsaufwand höher sein. Da die Wiederverwendbarkeit von Code jedoch auch bei anderen Datenstrukturen möglich wird, empfiehlt sich in diesem Kontext die Verwendung von Funktionen höherer Ordnung besonders.

3 Listenkomprehension

Einige der Konstrukte, die sich mit Higher-Order-Funktionen realisieren lassen, können in Haskell auch mit sogenannten *Listenkomprehensionen* realisiert werden. Listenkomprehensionen sind syntaktisch „schönere“ und klar leserliche Ausdrücke, mit denen eine Liste von bestimmten Elementen erstellt werden kann.

Die Syntax von Listenkomprehensionen ähnelt stark den aus der Mengenlehre bekannten Mengennotation. Hierbei ist zu beachten, dass Listen in Haskell im Gegensatz zu Mengen geordnet sind. Zudem können Elemente mehrfach vorkommen. Die Beispiele aus Abschnitt 1 zur Filterung einer Altersdatenliste nach Volljährigkeit und dem Mapping auf das Geburtsjahr lassen sich in der Mengenlehre folgendermaßen ausdrücken: $\{2019 - x \mid x \in \{2, 5, 18, 24, 4, 56\}, x \geq 18\}$ Die Umsetzung dieses Beispiels mit Listenkomprehensionen erfolgt ähnlich:

Listing 18:

```haskell
[2019 - x | x <- [2,5,18,24,4,56], x >= 18]
= [18,24,56]
```

Daraus lässt sich eine generelle Syntax ableiten, die der der Mengennotation sehr ähnlich ist:

```haskell
[Ausdruck | Qualifikatoren]
```

Jede Listenkomprehension besteht aus einem Ausdruck und mindestens einem *Qualifikator*. Qualifikatoren stellen die Variablen bereit, die im Ausdruck der Listenkomprehension verwendet werden können, oder schränken diese ein. Dabei wird zwischen zwei Arten von Qualifikatoren unterschieden: *Generatoren* und *Einschränkungen*. Jeder dieser Qualifikatoren wird mit einer eindeutigen Regel ausgewertet, welche sich auf Higher-Order-Funktionen zurückführen lässt. Damit ist eine Listenkomprehension lediglich eine syntaktisch elegante Kurzschreibweise für eine Verkettung von Higher-Order-Funktionen. Die Auswertung der Listenkomprehension bzw. der Qualifikatoren wird durch zwei Regeln - die *Generatorregel* und die *Einschränkungsregel* - formalisiert [Gie16].

Generatorregel Durch einen Generator kann eine Variable mit verschiedenen Werten belegt werden. Dabei wird die Syntax `v <- lst` benutzt, wobei `v` die zu belegende Variable und `lst` eine Liste bzw. ein Ausdruck, welcher eine Liste erzeugt, ist. Dabei wird wie folgt ersetzt:

Listing 19:

```
1 [expr | v <- lst, Q] = concat (map f lst) where f v = [expr | Q]
```

`Q` ist eine Liste mit Qualifikatoren, welche in den nächsten Auswertungssschritten ausgewertet werden. Für jedes Listenelement wird durch eine weitere Listenkomprehension, welche die auszuwertende Generatorregel nicht mehr enthält, eine Listenkomprehension erzeugt. In diesen ist jeweils die Variable `v` durch die Variablen aus `lst` ersetzt. Dies geschieht durch die Benutzung von `map`. Die darauf entstehenden Listen werden mit `concat` zusammengefügt [Gie16].

Einschränkungsregel Die Einschränkungsregel wird primär dazu verwendet Elemente der Liste zu filtern. Sie wird durch einen Ausdruck `e` angegeben, welcher zu einem Booleschen Wert auswertet. Dabei wird die Listenkomprehension wie folgt ersetzt:

Listing 20:

```
1 [expr | e, Q] = if e then [expr | Q] else []
```

`Q` beinhaltet wieder beliebig viele Qualifikatoren, welche weiterführend unter Verwendung der beiden Regeln ausgewertet werden. Wird durch die Funktion `True` zurückgegeben, so wird die Listenkomprehension weiter ausgewertet. Wird `False` zurückgegeben, so wird zur leeren Liste ausgewertet.

Durch diese beiden Auswertungsregeln kann eine syntaktisch korrekt angegebene Listenkomprehension ausgewertet werden. Bei jeder Anwendung einer Regel wird die Anzahl der Qualifikatoren um eins verringert. Sind alle Qualifikatoren ausgewertet - wenn also `Q` keine Qualifikatoren enthält -, so wird zu `[expr]` ausgewertet.

Die Auswertung von Listing 18 ist dann wie folgt:

```
1 [2019 - x | x <- [2,5,18,24,4,56], x >= 18]
2 = concat (map f [2,5,18,24,4,56]) where f x = [2019 - x | x >= 18]
3 = [2019 - 2| 2 >= 18] ++ [2019 - 5 | 5 >= 18] ++ [2019 - 18 | 18 >= 18] ++
4   [2019 - 24 | 24 >= 18] ++ [2019 - 4 | 4 >= 18] ++ [2019 - 56 | 56 >= 18]
5 = [] ++ [] ++ [2019 - 18] ++ [2019 - 24] ++ [] ++ [2019 - 56]
6 = [2019 - 18, 2019 - 24, 2019 - 56] = [2001,1995,1963]
```

4 Auswertungsstrategie

Haskell unterscheidet sich - als funktionale Programmiersprache - durch die Art der Auswertung stark zu klassischen imperativen bzw. objektorientierten Programmiersprachen. Im Fall von *Java* wird beispielsweise die *strikte Auswertung* verwendet, welche erst alle übergebenen Argumente von Funktionen komplett auswertet, bevor die eigentliche Funktion ausgewertet wird.

4.1 Lazy-Auswertungsstrategie

Die Auswertungsstrategie von Haskell ist die *Lazy Evaluation*, welche auf der *Leftmost-Outermost* Auswertungsstrategie basiert. Gegensätzlich zur strikten Auswertung wird dabei die Auswertung äußerer Funktionen priorisiert, bevor die Argumente ausgewertet werden. Die Ausdrücke

werden von links nach rechts ausgewertet. Ist es nach dieser Strategie nicht möglich einen Ausdruck auszuwerten, wenn beispielsweise kein Pattern beim *Pattern Matching* passend ist, so wertet Haskell den Ausdruck nur soweit aus, bis die äußere Funktion weiter ausgewertet werden kann. Dieses Prinzip wird in Verbindung mit Pattern Matching in Listing 21 deutlich. Das erste Argument, also `3 - 2`, muss ausgewertet werden, um festzustellen, ob der erste Parameter 0 ist. Erst danach kann entschieden werden, wie `add` ausgewertet wird. `7 + 3` muss jedoch nicht ausgewertet werden, da `add` bereits vorher ausgewertet werden kann [Gie16].

Listing 21:

```
1 add :: Int -> Int -> Int
2 add 0 y = y
3 add x y = x + y
4 Auswertung:
5 add (3 - 2) (7 + 3) = add 1 (7 + 3) = 1 + (7 + 3) = 1 + 10 = 11
```

Zusätzlich wertet Haskell bestimmte doppelte Ausdrücke nur einmal aus. Wird eine äußere Funktion ausgewertet und benötigt bei der Auswertung einen übergebenen Parameter mehrfach, so wird dieser nur einmal ausgewertet. Dieser Vorgang kann durch die Ersetzung mit `let`-Ausdrücken dargestellt werden.

Listing 22 beschreibt diesen Vorgang für die Funktion `double`. Nach der Leftmost-Outermost Auswertungsstrategie wird die Funktion `double` als erstes ausgewertet. `2 * 3` wird jedoch, damit keine Mehrfachberechnungen nötig sind, durch einen `let`-Ausdruck ersetzt [Bir15].

Listing 22:

```
1 double :: Int -> Int
2 double x = x + x
3 Auswertung:
4 double (2 * 3) = let x = 2 * 3 in x + x = let x = 6 in x + x = 6 + 6 = 12
```

Listing 23:

```
1 (2*3) + (2*3) = 6 + (2*3) = 6 + 6 = 12
```

Im Allgemeinen speichert Haskell jedoch nicht jedes Zwischenergebnis. In Listing 23 wird erneut $(2 \cdot 3) + (2 \cdot 3)$ berechnet, jedoch ohne die explizite Verwendung von `double`. Da der Compiler an dieser Stelle keine Verknüpfung zwischen den beiden identischen Teilausdrücken `2 * 3` herstellen kann, werden beide Ausdrücke gesondert berechnet [Bir15].

Durch die Kombination der beiden oben beschriebenen Paradigmen werden oft doppelte Berechnungen und Berechnungen, welche für das Ergebnis der Funktion nicht benötigt werden, vermieden.

Listing 24: Definition und Auswertung von head

```
1 head :: [a] -> a
2 head (x:_) = x
3 Auswertung:
4 head [1,double 1000000] = head 1:[double 1000000] = 1
```

Listing 24 zeigt diesen Vorteil auf. Durch `head` wird aus einer Liste beliebigen Typs das erste Element zurückgegeben. Wird die leere Liste übergeben, so gibt es einen Fehler. Die Berechnung der gesamten Liste ist für das Endergebnis nicht relevant. Durch die Auswertungsstrategie wird zusätzlich als erstes `head` ausgewertet, woraus sich bereits das Endergebnis ergibt. Eine

Auswertung von `double 1000000` ist aus diesem Grund nicht notwendig. Diese Ausführung ist im Vergleich zur strikten Auswertung, welche die Parameter zuerst auswertet, einfacher und ermöglicht es zeitintensive und redundante Berechnungen auszulassen [Bir15]. Erweitert ist es möglich Datenstrukturen beliebiger - sogar unendlicher - Größe zu erstellen, deren Auswertung mit strikter Auswertung nicht terminieren würde.

4.2 Programme auf unendlichen Listen

Durch die Lazy Evaluation lassen sich in Haskell Programme auf *unendlichen Datenstrukturen* erstellen. Ein solches Programm, welches beispielsweise aus einer unendlichen Liste endlich viele Elemente extrahiert, kann eine terminierende Ausgabe generieren. Beispiele für solche unendlichen Listen bzw. Folgen sind zum Beispiel die Fibonacci-Folge, die Folge aller Fakultäten von den natürlichen Zahlen oder sogar die natürlichen Zahlen selbst. Allgemeiner kann in Haskell eine aufsteigende unendliche Liste von ganzen Zahlen beginnend mit k durch `[k..]` erzeugt werden. Die Anzeige dieser Liste terminiert jedoch nicht.

Um terminierende Programme auf unendlichen Listen zu erstellen, müssen aus einer gegebenen unendlichen Liste endlich viele Werte extrahiert werden. Da Haskell die Lazy Evaluation verwendet, wird die unendliche Liste nur so weit ausgewertet wie es nötig ist, um das Ergebnis zurückzugeben. Die unendliche Folge der Fibonacci-Zahlen kann durch eine unendliche Liste dargestellt werden:

Listing 25: Fibonacci-Folge als unendliche Liste [Has14]

```
1 fib :: [Integer]
2 fib = map fibInner [0..]
3       where fibInner 0 = 0
4             fibInner 1 = 1
5             fibInner n = fib !! (n - 1) + fib !! (n - 2)
```

Es wird eine unendliche Liste von 0 aus aufsteigend erzeugt. Mittels der `map`-Funktion wird anschließend die im `where`-Teil definierte `fibInner`-Funktion auf die unendliche Liste abgebildet. Mit `!! (n - 1)` wird das $(n-1)$-te Listenelement, also $\text{fib}(n-1)$, gelesen und dann mit $\text{fib}(n-2)$ addiert, was $\text{fib}(n)$ ergibt. Dabei wird rekursiv wieder auf `fib` zugegriffen. Die Werte für 0 und 1 werden bei dieser Berechnung durch die `fibInner`-Funktion vorgegeben. Die Ausführung von `fib` terminiert nicht. Mit dem Listenzugriffs-Operator `!!`, `head` oder `take k` (siehe Listing 26), welches die ersten k Listenelemente als Liste zurückgibt, kann jedoch eine brauchbare Ausgabe generiert werden. Durch die Verwendung der Lazy Evaluation werden dabei nur die benötigten Werte ausgewertet.

Listing 26: Definition take[Gie16]

```
1 take :: Int -> [a] -> [a]
2 take _ [] = []
3 take n (x:xs) | n <= 0 = []
4               | otherwise = x : (take (n-1) xs)
```

Listing 27: Auswertung von take 3 fib

```
1 take 3 fib
2 = take 3 (fibInner 0:(map fibInner [1..]))
3 = fibInner 0:(take 2 (map fibInner [1..]))
4 = 0:(take 2 (map fibInner [1..]))
5 = 0:(take 2 (fibInner 1:map fibInner [2..]))
```

```
= 0:(fibInner 1):(take 1 (map fibInner [2..]))
= 0:1:(take 1 (map fibInner [2..]))
= 0:1:(take 1 (fibInner 2:map fibInner [3..]))
= 0:1:(take 1 (fibInner 2:map fibInner [3..]))
= 0:1:(fibInner 2):(take 0 (map fibInner [3..]))
= 0:1:(fib 1 + fib 0):(take 0 (map fibInner [3..]))
= 0:1:(0 + 1):(take 0 (map fibInner [3..]))
= 0:1:1:(take 0 (map fibInner [3..]))
= 0:1:1:[] = [0,1,1]
```

Listing 27 zeigt die Auswertungsschritte von `take 3 fib`. Durch das rekursiv definierte `take` werden die ersten drei Werte der Liste ausgegeben. Die Lazy Evalution macht es möglich, dass nur Teile der Liste ausgewertet werden. Eine komplette Auswertung wäre nicht möglich. Die `map`-Funktion muss jedoch teilweise ausgewertet werden, um zu überprüfen, ob die leere Liste zurückgegeben wird. Dies ist für das Pattern Matching der `take`-Funktion nötig. Nach der Berechnung der ersten drei Werte durch `map` terminiert das Programm und gibt das Ergebnis zurück.

Da `fib` nicht an ein Argument gebunden ist, sondern allgemeingültig verwendbar ist, wird es möglich, dass die unendliche Liste bzw. der Teil, welcher bereits berechnet ist im Speicher gehalten wird und somit für spätere Berechnungen zur Verfügung steht. Der *Glasgow Haskell Compiler* speichert `fib` zwischen. Die Funktion `fibInner` benutzt die vorher berechneten Zwischenergebnisse, welche nicht doppelt berechnet werden müssen. Dadurch wird die Anzahl an Rekursionsschritten und somit auch die Anzahl der Berechnungen generell verringert.

In Listing 28 ist die Fibonacci-Funktion aus Listing 25 leicht abgeändert angegeben. Es wird keine unendliche Liste mehr zurückgegeben, sondern für ein übergebenes Argument n der Funktionswert fib(n). Die unendliche Liste sowie die Ergebnisse werden hier jedoch nicht zwischengespeichert, da der Compiler nicht erkennen kann, dass die unendliche Liste nicht vom Parameter n abhängt, sodass bei erneuter Berechnung und bei der Berechnung der Zwischenwerte wieder alle Rekursionschritte durchlaufen werden müssen [Bir15].

Listing 28:

```
fibSlow :: Int -> Integer
fibSlow n = map fibInner [0..] !! n
      where fibInner 0 = 0
            fibInner 1 = 1
            fibInner n = fibSlow (n - 1) + fibSlow (n - 2)
```

Durch das Zusammenspiel der oben genannten Aspekte, ist es möglich, dass `fib 1000` in wenigen Millisekunden terminiert. Die Laufzeit ist deutlich geringer, da jede Fibonacci-Zahl nur einmal berechnet werden muss und dann in den folgenden Schritten wiederverwendet wird. Werden aufwendigere Berechnungen mehrfach ausgeführt, so ist die Liste bis zu dem benötigten Wert bereits berechnet und es ist nur der Listenzugriff nötig, welcher sehr schnell ist. Die Laufzeit von `fibSlow` ist hingegen exponentiell.

Zirkuläre Datenobjekte Eine besondere Art von unendlichen Datenstrukturen sind die *zirkulären Datenobjekte*. Sie besitzen die Eigenschaften von unendlichen Datenstrukturen, sind darüber hinaus jedoch auch noch bezüglich des Speichers optimiert. Bei Betrachtung von Listing 25 fällt auf, dass der Speicherverbrauch in $\Omega(n)$ liegt, wobei n die Anzahl der bereits berechneten Elemente der Liste ist. Es gibt jedoch Listen, welche sich wiederholen. Diese zirkulären

Datenobjekte besitzen einen Index $m \in \mathbb{N}_{>0}$, für den gilt:

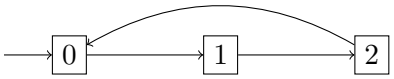
$$\text{Für alle } k \in \mathbb{N}_{>0}, i \in \mathbb{N} \text{ mit } 0 \leq i < m \text{ gilt: } \text{List}[i \cdot k] = \text{List}[i]$$

Die Liste wiederholt sich ab einem bestimmten Index m. Eine solche Liste wird vom Glasgow Haskell Compiler automatisch als zyklisch verkettete Liste gespeichert, in der der Pointer auf den Nachfolger des m-ten Elemente wieder auf den Head der Liste verweist [Gie16].
In Listing 29 und Abbildung 1 ist eine solches Datenobjekt modellhaft angegeben.

Listing 29:

```
1 mod3 :: [Int]
2 mod3 = 0:1:2:mod3
```

Abbildung 1: Listenstruktur:



In Abbildung 1 ist zu erkennen, dass der Speicherbedarf beim Beispiel nur in $\mathcal{O}(3)$ und allgemein in $\mathcal{O}(m)$ liegt.

Der praktische Nutzen dieser Optimierung durch den Compiler liegt in der Verringerung des Speicherbedarfs. Dieses Beispiel enthält als i-tes Listenelement $i \mod 3$. So könnte zum Beispiel ein Restklassenring bzw. -körper realisiert werden. Zugriff auf das Listenelement x Elemente weiter entspricht so einer Addition mit x.

5 Zusammenfassung

Zusammenfassend ist die Verwendung von Higher-Order-Funktionen in Kombination mit intelligenter Ausnutzung der Auswertungsstrategie in Haskell ein mächtiges Werkzeug, durch welches komplexere Probleme auf wenige Zeilen Code reduziert werden können. Dadurch besteht die Möglichkeit einfachen, lesbaren und effizienten Code zu schreiben, welcher insbesondere gut wiederverwendbar ist. Die Listenkomprehension, welche sich direkt aus den Higher-Order-Funktionen ableiten lässt, bietet dazu eine simple, aber gleichzeitig mächtige Möglichkeit mit relevanten Daten befüllte Listen aufzubauen.

Die genauen Hintergründe der Auswertung der `fib`-Funktion aus Listing 25 liegen in der Funktionsweise des Haskell-Compilers. Eine detailliertere Erläuterung würde den Rahmen dieser Arbeit übersteigen. Haskell bietet darüber hinaus noch weitaus mehr Möglichkeiten zur Programmoptimierung. Nicht behandelt wurden in dieser Arbeit beispielsweise strikte Funktionen, welche es dem Programmierer ermöglichen die Auswertung eines Argumentes zu erzwingen, obwohl dies durch die Lazy Evaluation nicht passiert wäre. Für einen detaillierteren Blick sei auf die referenzierte Literatur verwiesen.

Literatur

[Has14] HaskellWiki. *Memoization — HaskellWiki*. [Online; accessed 2-May-2019]. 2014. URL: `https://wiki.haskell.org/index.php?title=Memoization&oldid=57978`.

[Bir15] Richard Bird. *Thinking Functionally with Haskell*. 2015.

[Gie16] Jürgen Giesl. *Funktionale Programmierung; Vorlesungsskript*. 2016.