Skriptteil zur Vorlesung:
Proinformatik - Funktionale Programmierung

Dr. Marco Block-Berlitz

22.Juli 2009

Einfache Datenstrukturen

Durch geeignete Kombinationen aus den Basisdatentypen lassen sich neue, komplexere Strukturen von Daten erstellen. In diesem Kapitel werden Listen vorgestellt. Listen sind dabei geordnete Speicher für Elemente gleichen Typs. Der Umgang mit größeren Datenmengen wird dabei erleichtert und eine höhere Datenabstraktion ist dadurch möglich.

Da bei den Listen aber die Bedingung gilt, dass alle Elemente einer Liste vom gleichen Typ sein müssen, werden in diesem Kapitel auch Tupel vorgestellt. Die Typen der Elemente dürfen sich dabei unterscheiden, aber die Anzahl der Elemente ist fest.

Listen

Ein wichtiges Konzept zur Speicherung von mehreren Daten des gleichen Typs stellen Listen dar. Eine Liste ist dabei eine geordnete Menge von Elementen gleichen Typs. Listen werden in eckigen Klammern geschrieben und die Elemente innerhalb einer Liste mit Kommata getrennt. Beispiel einer Liste, dessen Elemente vom Typ `Int` sind

```
[1,2,3,4,5]
```

Eine Liste kann auch leer sein und somit keine Elemente enthalten

```
[]
```

oder andere Elementtypen verwalten. So beispielsweise eine Liste von Zeichen

```
['h','a','l','l','o']
```

Eine Liste von Zeichen kann auch vereinfacht in der Notation für Zeichenketten

```
"hallo"
```

angegeben werden. Ein Test auf der Eingabekonsole soll die Gleichheit beider Varianten überprüfen

```
Hugs> "hallo" == ['h','a','l','l','o']
Hugs> True
```

Es lassen sich auch Listen von Listen definieren, falls diese Listen ebenfalls den gleichen Typ besitzen

```
[[1,2], [5,-1], []]
```

Die folgende Liste ist demnach nicht erlaubt

```
Hugs> [[2],['d','e']]
```

und führt auch prompt zu einem Fehler

```
Hugs> ERROR - Cannot infer instance
Hugs> *** Instance   : Num Char
Hugs> *** Expression : [[2],['d','e']]
```

Zerlegung in Kopf und Rest

In Haskell gibt es eigentlich nur eine Methode, um Elemente aus einer Liste herauszunehmen. Alle weiteren, die noch in diesem Skript vorgestellt werden, lassen sich auf diese zurückführen. Mit dem `(:)`-Operator wird eine Liste zerlegt in Kopf und Rest. Der Kopf ist dabei das erste, ganz links stehende Element.

$$x_1 : [x_2, x_3, ..., x_n]$$

Gleichzeitig läßt sich auf diese Weise das Einfügen eines Elementes in eine Liste formulieren.

Die beiden Notationen `[1,2,3]` und `1:[2,3]` repräsentieren beide die Liste mit den Elementen $1, 2$ und 3. Der Vorteil der `(:)`-Darstellung ist in der Manipulation von Listen begründet. So läßt sich jede n-elementige Liste als das n-fache Einfügen der Elemente in eine leere Liste darstellen

$$x_1 : (x_2 : (x_3 : (... : (x_n : [])...)))$$

Die Klammern dürfen in diesem Fall sogar weggelassen werden, die Begründung dazu folgt in einem späteren Kapitel. Im Folgenden werden ein paar Beispiele vorgestellt

```
Hugs> 'a':['b','c']
Hugs> ['a','b','c']

Hugs> 1:2:2:[]
Hugs> [1,2,2]

Hugs> [1,2]:[]
Hugs> [[1,2]]
```

Das Kopf-Rest-Prinzip ist sehr wichtig und findet sehr häufig Verwendung. Die Rückgabe des ersten Elements einer Liste läßt sich in einer Funktion ausdrücken

```
erstes :: [a] -> a
erstes (x:_)   = x
```

Dazu wird der `(:)`-Operator verwendet, um die Eingabeliste beliebigen Typs zu zerlegen. Der Zugriff der Funktion `erstes` auf eine leere Liste führt zu einem Fehler

```
Hugs> erstes []
Program error: pattern match failure: erstes []
```

Für alle anderen Listen liefert die Funktion erfolgreich das erste Element zurück

```
Hugs> erstes [1]
Hugs> 1

Hugs> erstes "hallo"
Hugs> 'h'
```

In Haskell gibt es bereits die vordefinierte Funktion `head`, die diese Funktionalität anbietet. Da der `(:)`-Operator eine Liste in zwei Teile zerlegt, läßt sich auch die restliche Liste ohne das erste Element als Funktion angeben.

```
rest :: [a] -> [a]
rest (_:xs)    = xs
```

Der Datentyp des Rückgabewerts `xs` ist demzufolge nicht `a` sondern eine Liste vom Typ `a`. Ein kurzes Beispiel zeigt die Verwendung der Funktion

```
Hugs> rest [1]
Hugs> []

Hugs> rest "hallo"
Hugs> "allo"
```

Die in Haskell implementierte Standardfunktion dazu heißt `tail`.

Rekursive Listenfunktionen

Um die beiden umgekehrten Fälle in Funktionen ausdrücken zu können muss auf Rekursion zurück gegriffen werden. Eine Funktion, die das letzte Element zurückliefern soll, durchläuft zunächst alle Elemente bis eine einelementige Liste übrigbleibt und gibt das vorhandenen Element zurück.

```
letztes :: [a] -> a
letztes [x]    = x
letztes (_:xs) = letztes xs
```

Für den anderen Fall werden die Elemente solange in einer neuen Liste gesammelt, bis die einelementige Liste erreicht ist, diese wird dann durch eine leere ersetzt und somit das letzte Element entfernt.

```
ohneletztes :: [a] -> [a]
ohneletztes [x]    = []
ohneletztes (x:xs) = x:ohneletztes xs
```

Die beiden vordefinierten Haskellfunktionen für `letztes` und `ohneletztes` heißen `last` und `init`. Die folgende Funktion überprüft, ob ein Element in einer Liste enthalten ist[1].

```
enthalten [] _      = False
enthalten (x:xs) y = (x==y) || enthalten xs y
```

Auch diese Funktion ist bereits in Haskell mit der Bezeichnung `elem` definiert.

[1] Die Signatur der Funktion wird in einem der späteren Kapitel vorgestellt, da dafür ein weiteres Konzept benötigt wird.

Zusammenfassen von Listen

Listen, die den gleichen Typ aufweisen, können zu einer Liste zusammengefasst werden. Dabei stehen die Elemente der ersten Liste vor denen der zweiten. Als Operator für das Zusammenfassen (Konkatenieren) von Listen kann der (++)-Operator verwendet werden.

```
Hugs> [1,2] ++ [3,4]
Hugs> [1,2,3,4]
```

Es folgen ein paar kleine Beispiele für die Anwendung der Konkatenation

```
Hugs> [] ++ ['a','b','c']
Hugs> ['a','b','c']

Hugs> ['1','0'] ++ []
Hugs> ['1','0']

Hugs> ([1,2]++[3,4])++[5,6] == [1,2]++([3,4]++[5,6])
Hugs> True
```

Eine Funktion `fassezusammen`, die zwei Listen analog zum (++)-Operator konkateniert läßt sich wie folgt rekursiv über den (:)-Operator definieren

```
fassezusammen :: [a] -> [a] -> [a]
fassezusammen []     ys = ys
fassezusammen (x:xs) ys = x:fassezusammen xs ys
```

Ein kleiner Funktionstest dazu

```
Hugs> fassezusammen [1,2] [3,4]
Hugs> [1,2,3,4]
```

Automatische Erzeugung von Listen

Listen lassen sich in Haskell alternativ auch in einer kompakten Form automatisch generieren (*list comprehensions*). Eine Analogie zur mathematischen Notation von Mengen ist durchaus gegeben und gewünscht. In der Mathematik läßt sich die Menge der geraden, natürlichen Zahlen beispielsweise so beschreiben

$$\{x \,|\, x \in \mathbb{N} \wedge x\, ist\, gerade\, Zahl\}$$

Bei der automatischen Erzeugung von Listen in Haskell sieht der Mechanismus ähnlich aus. Zunächst ein einfaches Beispiel

```
[a | a<-[1,2,3,4,5]]
```

Es werden dabei alle Elemente `a` aus der Liste `[1,2,3,4,5]` von links nach rechts durchgegangen und da keine weiteren Bedingungen gelten sofort in die neue Liste übernommen. Nach Ausführung dieser Zeile in die Haskell-Konsole ergäbe das die Originalliste

```
Hugs> [a | a<-[1,2,3,4,5]]
Hugs> [1,2,3,4,5]
```

Beim Durchlaufen der Liste über den Parameter `a`, lassen sich jetzt Bedingungen angeben. Falls diese Bedingungen erfüllt sind, wird der Inhalt von `a` in die neue Liste übernommen. Beispielsweise könnten nur die geraden Zahlen von Interesse sein

```
[a | a<-[1,2,3,4,5], mod a 2==0]
```

Das würde die neue Liste mit zwei Elementen

```
Hugs> [a | a<-[1,2,3,4,5], mod a 2==0]
Hugs> [2,4]
```

erzeugen. Da eine Bedingung einen Wahrheitswert liefert, kommen als Bedingungen nur Boolesche Funktionen in Frage. Im ersten Beispiel wurde dabei keine Bedingung verwendet. Es lassen sich aber mehr als nur eine Bedingung angeben

```
Hugs> [a | a<-[0,1,2,3,4,5], mod a 2/=0, a/=1]
Hugs> [3,5]
```

Es wird die Liste der ungeraden Elemente ohne 1 aus der Liste `[0,1,2,3,4,5]` erzeugt.

Automatisches Aufzählen von Elementen

Listen lassen sich auch durch Startwert, Schrittweite und Zielwert konstruieren

```
[s1, s2 .. sn]
```

Die ersten beiden Elemente liefern den Startwert `s1` und die Schrittweite `s2-s1`. Das erste Element `k1` der neuen Liste ist `s1`, das nächste Element `k2` ergibt sich aus `s1+(s2-s1)` ist also `s2`. Darauf folgt `s2+(s2-s1)` usw. Es können solange neue Elemente in die neue Liste eingefügt werden[2], bis zum letzten Element `km`, mit $m = 0.5 + \lfloor \frac{s_n}{Schrittweite} \rfloor$. Eine genaue Beschreibung ist dem Haskell-Report 98 zu entnehmen (siehe dort unter „Arithmetic Sequences" [10]).

Ein Beispiel dazu

```
[1, 2 .. 10]
```

Der Startwert ist 1 und die Schrittweite +1. Die so erzeugte Liste hat die Werte

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Die folgenden beiden Darstellungen kombinieren das Kopf-Rest-Prinzip mit der Listenerzeugung und liefern ebenfalls die Elemente 1 bis 10

```
1:[2, 3 .. 10]
```

Falls nur ein Element als Startelement angegeben wird, so wird die Schrittweite automatisch auf +1 gesetzt

[2] An dieser Stelle sei angemerkt, dass die Konvention, dass das letzte Element auch gleichzeitig eine obere Schranke darstellt, weder bei Hugs noch GHC erfüllt wird.

```
1:2:[3 .. 10]
```

Die Schrittweite läßt sich auch varrieren, schauen wir uns ein paar Beispiele an

```
Hugs> [2, 4 .. 10]
Hugs> [2,4,6,8,10]

Hugs> [0,-1 .. -10]
Hugs> [0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]

Hugs> [3.1 .. 6.5]
Hugs> [3.1,4.1,5.1,6.1]

Hugs> [3.1 .. 6.6]
Hugs> [3.1,4.1,5.1,6.1,7.1]
```

Wir wollen die Anzahl der Elemente einer Liste durch die Funktion `laengeListe` rekursiv bestimmen:

```
laengeListe :: [Int] -> Int
laengeListe []     = 0
laengeListe (_:xs) = 1 + laengeListe xs
```

Testen wir die Funktion

```
Hugs> [3 .. 6]
[3,4,5,6]

Hugs> laengeListe [3 .. 6]
4
```

Unendliche Listen

Haskell erlaubt auch das Arbeiten mit unendlichen Listen. Wenn das Intervallende nicht angegeben wird, erzeugt Haskell je nach Bedarf und Speicherplatz neue Elemente.

Als Beispiel wollen wir die Liste aller Quadrate aus natürlichen Zahlen definieren

```
quadrate :: [Int]
quadrate = [n*n | n <- [0 ..]]
```

Das Ausführen führt nach der Ausgabe einer Vielzahl von Quadratzahlen zu einem Speicherfehler. Das wir eine unendliche Liste nicht komplett anzeigen lassen können, sollte uns nicht entmutigen, das haben wir erwartet.

Haskell zeigt aber mit dem Einsatz von unendlichen Listen eine besondere Eigenschaft. So werden nur die Elemente einer unendlichen Liste ausgewertet, die auch wirklich benötigt werden. Das macht Haskell zu einer mächtigen Sprache. Schauen wir uns als Beispiel die Primzahlen an.

Algorithmus von Eratosthenes

Eine Primzahl ist nur durch sich selbst und durch 1 teilbar und hat genau zwei Teiler. Demzufolge ist 2 die kleinste Primzahl. Mit dem Algorithmus von Eratosthenes lassen sich Primzahlen kleiner oder gleich einer vorgegebenen Obergrenze erzeugen.

Für unser folgendes Beispiel sollen alle Primzahlen bis 40 erzeugt werden. Dazu notieren wir uns zunächst alle Elemente von 2 bis 40 in einer Liste.

~~1~~	**2**	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40

Die 2 als erstes Element der Liste stellt eine Primzahl dar. Die grauen Zahlen sind noch Primzahlkandidaten. Wir können die 2 als Primzahl markieren und streichen alle Vielfachen von 2 aus der restlichen Liste (rot und durchgestrichen).

~~1~~	**2**	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~

Die 3 als nächstes Element erfüllt wieder die Primzahleigenschaften.

~~1~~	**2**	**3**	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~

Anschließend werden alle Vielfachen von 3 gestrichen.

~~1~~	**2**	**3**	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~

Das wird solange vorgenommen, bis die Liste der noch zur Verfügung stehenden Elemente leer ist.

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~

Daraus ergibt sich die Liste der Primzahlen $[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]$ kleiner/gleich 40.

In Haskell läßt sich ein Algorithmus sehr einfach ableiten, der zwar zunächst dem Sieb des Eratosthenes ähnlich sieht, aber nicht genau die Vorgehensweise aufweist. Es sei hier nur am Rande erwähnt, dass sich genau das erheblich auf die Laufzeit auswirkt und bei Melissa E. O'Neill bereits ausführlich diskutiert wurde [1]. Zu einem späteren Zeitpunkt werden wir darauf noch einmal zusprechen kommen.

```
primzahlen = sieb [2 ..]
   where sieb (x:xs) = x:sieb [n|n<-xs, mod n x > 0]
```

In der lokal definierten Funktion `sieb` werden die Vielfachen der ersten Primzahlkandidaten aus der neuen Liste entfernt. Da Haskell aber nur die Auswertungen ausführt, die notwendig sind, um das nächste Element der Liste zu bestimmen, läßt sich die Liste der Primzahlen ausgeben, bis der Speicherplatz nicht mehr ausreicht[3] .

```
Hugs> primzahlen
Hugs> [  2,  3,  5,  7,  11,  13,  17,  19,
Hugs>   23, 29, 31, 37,  41,  43,  47,  53,
Hugs>   59, 61, 67, 71,  73,  79,  83,  89,
Hugs>   97, ... usw.
```

Listen zerlegen

Es kann sinnvoll sein, eine Liste mit m Elementen in zwei Teillisten zu zerlegen, wobei die ersten n Elemente in der einen und die restlichen Elemente in einer anderen Funktion ermittelt werden. Um die ersten n Elemente einer Liste zu erhalten, muss der Counter n nur bei jedem Rekursionsschritt dekrementiert werden. Die Funktion `nimm :: Int -> [a] -> a`

```
nimm :: Int -> [a] -> [a]
nimm _ []     = []
nimm 0 _      = []
nimm n (x:xs) = x:nimm (n-1) xs
```

Sollte $m < n$ sein, so werden einfach die ersten m Elemente zurückgeliefert

```
Hugs> nimm 2 [1,2,3,4]
Hugs> [1,2]

Hugs> nimm 5 [1,2]
Hugs> [1,2]
```

[3] Die folgende Ausgabe wurde auf Grund der Übersichtlichkeit etwas angepasst.

Um die restlichen Elemente zu erhalten, die nicht durch die Funktion `nimm` geliefert werden, definieren wir die Funktion `nimmnicht :: Int -> [a] -> [a]`

```
nimmnicht :: Int -> [a] -> [a]
nimmnicht 0 xs     = xs
nimmnicht _ []     = []
nimmnicht n (_:xs) = nimmnicht (n-1) xs
```

Schauen wir uns auch dafür ein Beispiel an

```
Hugs> nimmnicht 2 [1,2,3,4]
Hugs> [3,4]
```

Beide Funktionen werden auch in der prelude angeboten, dort heißen sie `take, drop :: Int -> [a] -> [a]`.

Tupel

Listen ähneln eher Mengen von Elementen. Bei Tupeln allerdings ist die Anzahl der Elemente eines Tupels und deren Position fest vorgegeben. Im Gegensatz zu Listen, lassen sich auch unterschiedliche Datentypen in Tupeln repräsentieren.

Schauen wir uns dazu ein Beispiel an

$$(x_1, x_2, x_3, ..., x_n)$$

Die Datentypen der einzelnen Elemente können sich unterscheiden, so z.B.

```
("Peter", 44)
```

An der ersten Position des zweistelligen Tupels steht ein `String` und an der zweiten Position ein `Int`.

Ein solches Tupel kann als Eingabe erwartet werden. Das folgende Beispiel zeigt die Zerlegung eines Tupels in die einzelnen Bestandteile

```
erstesElement :: (a, b) -> a
erstesElement (x, _) = x
```

Haskell erwartet jetzt für die Funktion `erstesElement` ein zweistelliges Tupel als Eingabe und liefert das erste Element zurück

```
Hugs> erstesElement ("Hans", 1)
Hugs> "Hans"
```

Analog dazu können auch die anderen Elemente zurückgeliefert werden. Im folgenden Beispiel wird ein dreistelligen Tupel in die Funktion `drittesElement` übergeben und der dritte Wert zurückgeliefert.

```
drittesElement :: (a, b, c) -> c
drittesElement (_, _, z) = z
```

Es gibt bereits die vordefinierten Funktionen `fst` und `snd`, die das erste bzw. zweite Element liefern.

Beispiel pythagoräisches Zahlentripel

Als pythagoräisches Zahlentripel werden drei natürliche Zahlen x, y und z bezeichnet, die die Gleichung $x^2 + y^2 = z^2$ erfüllen, mit $x < y < z$. Lösungen zu diesem Problem wurden bereits schon $1829 - 1500$ Jahre v. Chr. diskutiert [19].

In Haskell läßt sich das Problem direkt formulieren

```
pyT n = [(x,y,z) | x<-[1..n-2], y<-[x+1 .. n-1],
                   z<-[y+1 .. n], x*x+y*y==z*z]
```

Die Lösungen bis zu einem Maximalwert von `n=10` lassen sich so einfach ausgeben

```
Hugs> pyT 10
[(3,4,5),(6,8,10)]
```

Beispiel n-Dameproblem

Ein beliebtes Problem aus dem Spielebereich ist das 8-Dameproblem. Es werden dabei acht Damen so auf ein 8×8-Brett positioniert, dass diese sich nicht gegenseitig schlagen können. Eine Dame zieht dabei horizontal, vertikal und diagonal bis zum letzten Feld am jeweiligen Brettrand.

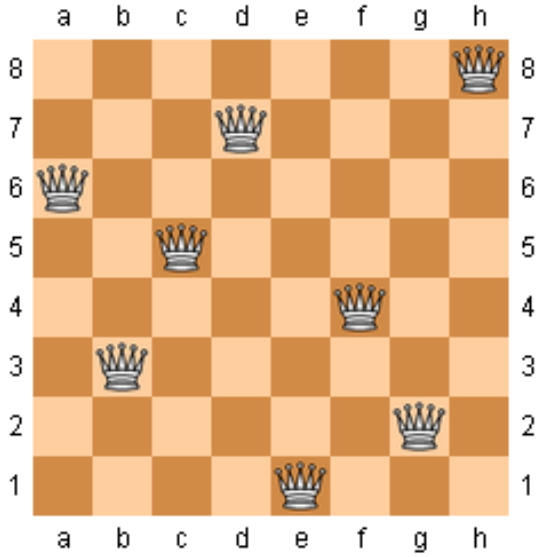

Abbildung 1: Eine mögliche Lösung des 8-Dameproblems (Abbildung aus [19]).

In Haskell läßt sich eine Lösung sehr elegant realisieren. Dabei lösen wir nicht nur das 8-Dameproblem, sondern gleich das n-Dameproblem auf einem $n \times n$ Schachbrett. Es läßt sich leicht überlegen, warum mindestens ein $n \times n$ Brett benötigt wird, um n Damen (die sich nicht schlagen können) zu beherbergen.

Wir wollen es so rekursiv lösen, dass wir alle möglichen Kombinationen erhalten. Wir schreiben eine Funktion `n_damen`, die n Damen auf dem Brett platzieren möchte und alle Lösungen als Listen von Positionen auf dem Feld zurückliefert.

Dafür formulieren wir zunächst die kleinste Problemstellung mit 0 Damen. Die Liste der möglichen Positionen, wobei eine Position ein Tupel bestehend aus x- und y-Komponente ist, ist für 0 Damen leer. Für n Damen gehen wir davon aus, dass wir die $n - 1$ Damen bereits auf $n - 1$ Spalten korrekt verteilt haben und in der letzten Spalte nur noch die letzte Dame einfügen müssen. Schauen wir uns die komplette Lösung an

```
n_damen :: Int -> [[(Int, Int)]]
n_damen max_n = damen max_n
  where damen n
    | n == 0    = [[]]
    | otherwise  = [posi++[(n,m)] | posi<-damen (n-1),
                                    m<-[1..max_n],
                                    sicher posi (n,m)]

  sicher list neu   = and [not(bed dame neu) | dame<-list]
  bed (i, j) (m, n) = (j==n) || (i+j==m+n) || (i-j==m-n)
```

Die rekursive Hilfsfunktion damen erhält mit `max_n` die maximale Anzahl Damen, die zu setzen ist. Durch den Aufruf der lokalen Funktion `damen n` behalten wir Zugriff auf die Variable `max_n` und können diese in der Funktionsdefinition verwenden. Falls `n=0` ist, wird eine Liste mit einer leeren Liste zurückgeliefert. Ist diese nicht 0, so wird mit Hilfe der automatischen Listenerzeugung (siehe entsprechenden Abschnitt) eine rekursive Beschreibung der Lösung erzeugt. In `posi` stehen die Positionen der `n-1` Damen, die bereits ohne gegenseitige Drohungen auf dem Feld stehen, hier findet quasi der Rekursionaufruf statt, denn zunächst werden `n-1` Damen plaziert, davor `n-2` und so fort. Dabei bezieht sich `n` auf die Zeile. Mit `m` testen wir für alle zur Verfügung stehenden Spalten, ob an der Position `(n,m)` eine neue Dame gesetzt werden kann. Die Funktion `sicher` übernimmt diese Aufgabe.

In `sicher` werden alle Damen, die bereits positioniert sind, mit der neuen Position verglichen. Sollte die neue Position im Bewegungsareal einer der Damen liegen, bricht die Funktion mit `False` ab. Über die `And`-Faltung der booleschen Resultate der Funktion `bed` wird das realisiert. Nur wenn alle Damen mit der neuen Position einverstanden sind, wird diese Position akzeptiert.

Der Algorithmus ist dann beendet, wenn alle `n` Damen regelkonform und frei von gegenseitigen Drohungen auf dem `n`×`n`-Feld positioniert sind.

Literatur

[1] O'Neill M.E.: „*The Genuine Sieve of Eratosthenes*", unpublished draft, siehe: *http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf* 8

[2] Block M.: „*Java Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*", Springer-Verlag 2007

[3] Dankmeier D.: „*Grundkurs Codierung: Verschlüsselung, Kompression, Fehlerbeseitigung.*", 3.Auflage, Vieweg-Verlag, 2006

[4] Schulz R.-H.: „*Codierungstheorie: Eine Einführung*", 2.Auflage, Vieweg+Teubner, 2003

[5] Schöning U.: „*Algorithmik*", ISBN-13: 978-3827410924, Spektrum Akademischer Verlag, 2001

[6] Pope B.: „*A tour of the Haskell Prelude*", unpublished (*http://www.cs.mu.oz.au/~bjpop/*), 2001

[7] Hudak P., Peterson J., Fasel J.: „*A gentle introduction to Haskell Version 98*", unpublished (*http://www.haskell.org/tutorial/*), 2000

[8] Cormen T.H., Leiserson C.E., Rivest R.L.: „*Introduction to Algorithms*", MIT-Press, 2000

[9] Gibbons J., Jones G.: „*The Under-Appreciated Unfold*", Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 273-279, United States, 1998

[10] Haskell-Onlinereport 98: *http://haskell.org/onlinereport/index.html* 5

[11] Webseite der Helium-IDE: *http://www.cs.uu.nl/wiki/Helium*

[12] Wikibook zur Datenkompression: *http://de.wikibooks.org/wiki/Datenkompression*

[13] Haskell-Funktionen: *http://www.zvon.org/other/haskell/Outputglobal/index.html*

[14] Webseite des Euler-Projekts: *http://projecteuler.net/*

[15] Webseite Haskellprojekte: *http://hackage.haskell.org/packages/archive/pkg-list.html*

[16] Projektwebseite Frag: *http://haskell.org/haskellwiki/Frag*

[17] Projektwebseite Monadius: *http://www.geocities.jp/takascience/haskell/monadius_en.html*

[18] Haskell-Suchmaschine Hoogle: *http://www.haskell.org/hoogle/*

[19] Wikipedia: *http://www.wikipedia.com* 10