www.haskell.org

Haskell, ein gutes Werkzeug der Diskreten Mathematik

von Jürgen Bokowski

Die Programmiersprache Haskell ist nach dem amerikanischen Mathematiker Haskell Brooks Curry (1900–1982) benannt, der 1929 bei Hilbert mit der Arbeit Grundlagen der kombinatorischen Logik *promovierte. Seine Arbeiten waren grundlegend für die Entwicklung funktionaler Programmiersprachen.*

Ich möchte in diesem Artikel auf die funktionale Programmiersprache Haskell 98 hinweisen und damit diejenigen ansprechen, die noch nie von ihr gehört haben oder sie noch nicht schätzen gelernt haben. Wie mir unlängst ein Informatiker schrieb, trifft dies ironischerweise gerade auf viele Mathematiker in Entscheidungspositionen zu. Was nichts kostet kann aber doch wertvoll sein.[1]

Ich wende mich insbesondere an Leser mit einem Bezug zur diskreten Mathematik, die bisher nur imperative Sprachen kennen, wie z. B. Fortran, Pascal, C, C^{++} oder Java. Nach dem Entstehen der Sprache Lisp blieb die Entwicklung der Informatik in der Theorie der funktionalen Sprachen keinesfalls stehen.

Der Widerstand des Lesers mag groß sein, sich mit einer weiteren Sprache zu beschäftigen, zumal nach Erreichen einer gewissen Altersgrenze. Haskell kommt jedoch der mathematischen Denkweise so nahe, dass dieses Argument hier eher wegfällt. Wir können einfache Funktionen und damit zugehörige Programme oft unmittelbar (ohne Einführung in Haskell) mit elementarer Schulmathematik verstehen. Das rechtfertigt den Einsatz in der Lehre. Wenn wir bescheiden starten und nicht sofort die letzten Feinheiten der Sprache Haskell 98 ansprechen, können wir nach kurzer Einarbeitung mit dem Verständnis unserer mathematischen Funktionen gleich deren rechnerische Auswertung miterhalten. Mathematische Funktionen beschreiben bereits das Programm.

Funktionale Programmierung benutzt mathematische Funktionen. Eine funktionale Sprache ist wesentlich verschieden von den oben genannten imperativen Sprachen. Informatiker und Ingenieure aus dem Bereich der Softwaretechnik empfehlen oft, ein Problem zunächst mit Hilfe einer funktionalen Programmiersprache zu modellieren, z. B. mit der de facto Standardsprache Haskell 98, bevor das Problem z. B. in C oder Java implementiert wird. Die gesamte Entwicklungszeit von der ersten Problemstellung bis zur fertigen Implementierung über eventuelle Modifikationen an der Problemstellung und alle Fehlersuchzeiten ist dann in der Regel bedeutend kürzer. Die Programme sind auch nach längerer Zeit noch verstehbar. Viele Kenner sowohl imperativer als auch funktionaler Programmiersprachen berichten übereinstimmend, dass erst nach eigenen Erfahrungen wirklich die Stärke von Haskell für mathematische Anwendungen erkennbar und geschätzt wird. Der Code ist kompakt, die Fehlersuche ist extrem kurz. Diese Erkenntnis hat sowohl unsere Mathematik-Fachbereiche als auch unsere Informatik-Fachbereiche noch nicht in voller Konsequenz erreicht. Andernfalls würde Haskell 98 wohl häufiger als erste Programmiersprache im ersten Semester in den Lehrplänen auftauchen. Zunächst müssen offenbar erst die Entscheidungsträger diese Erkenntnis gewinnen. Dazu könnte dieser Artikel beitragen.

Betrachten wir als erstes Beispiel die Pythagoräischen Tripel mit Komponenten zwischen 1 und n. Wir schreiben eine Funktion, die zu gegebener ganzen Zahl n diese Tripel liefert:

```
pythtripel::Integer->[[Integer]]
pythtripel n = [[a,b,c]|a<-s,b<-s,c<-s, a*a+b*b==c*c]
               where s = [1..n]
```

1 Haskell 98 ist akademische Freeware, `www.haskell.org`.

Die Haskell-Syntax ist eng mit der mathematischen Schreibweise verbunden. Die Funktion `pythtripel` ist unmittelbar auf Schulniveau verständlich. Diese Funktion mit Definitions- und Bildbereich ist bereits das fertige Programm. Dieses Geschenk einer 30-jährigen Entwicklung in der Informatik an die Mathematik sollten wir als Mathematiker freudig begrüßen.

Im zweiten Beispiel wählen wir die Fakultät als Funktion auf der Menge der natürlichen Zahlen. Die folgende rekursive Definition der Funktion ist wieder gleichzeitig das fertige Programm.

```
fak:: Integer->Integer
fak 0 = 1
fak n = n * (fak (n-1))
```

Die Stärke von Haskell zeigt sich besonders bei rekursiven Definitionen.

Wir wählen ein drittes Beispiel. Wir bestimmen die r-Tupel für ganzzahliges r aus einer Liste ganzer Zahlen. Einem Paar bestehend aus der Zahl r und einer Liste ganzer Zahlen soll damit eine Liste von Listen zugeordnet werden. Beachten Sie die entsprechende Funktionsschreibweise im anschliessenden Code. Für $r = 1$ ist das Ergebnis eine Liste mit ein-elementigen Listen. Hat die Liste r Elemente, dann besteht das Ergebnis nur aus einer Liste mit der gegebenen Liste als Element. Sonst betrachtet man zunächst die r-Tupel, die mit dem Kopfelement x der Liste beginnen und danach ein $(r-1)$-Tupel aus dem Rest der Liste. Dann fügt man die r-Tupel hinzu, die aus dem Rest der Liste gebildet werden können. Also alle r-Tupel, die das Kopfelement x nicht an erster Stelle besitzen. Die Funktion `tupel` kann wieder rekursiv verwendet werden. Im ersten Fall hat sich die Anzahl der Elemente eines Tupels verkleinert, im zweiten Fall enthält die Liste ein Element weniger.

```
tupel::Int->[Int]->[[Int]]
tupel   r     l
 |r == 1        = [[el] | el<-l ]
 |length l == r = [l]
 |otherwise     = (map([head l]++)(tupel(r-1)(tail l)))
                                ++  tupel  r  (tail l)
```

Die kompakte Anschrift des obigen Sachverhalts ist der interpretierbare Programmcode.

Nach diesen Beispielen zitiere ich zur Entwicklung der Sprache aus dem Vorwort zu [5].

> In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages. This book describes the result of that committee's efforts: a purely functional programming language called Haskell, named after the logician Haskell B. Curry whose work provides the logical basis for much of ours.
>
> **Goals**
>
> The committee's primary goal was to design a language that satisfied these constraints:
>
> 1. It should be suitable for teaching, research, and applications, including building large systems.
> 2. It should be completely described via the publication of a formal syntax and semantics.
> 3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
> 4. It should be based on ideas that enjoy a wide consensus.
> 5. It should reduce unnecessary diversity in functional programming languages.
>
> ... Haskell 98 provides a stable point of reference so that those who wish to write text books, or use Haskell for teaching, can do so in the knowledge that Haskell 98 will continue to exist.

Das Buch von Cordelia Hall und John O'Donnell, [3], benutzt nach 30 Seiten Einführung in die funktionale Programmiersprache Haskell auf weiteren 300 Seiten die Sprache zur Erklärung der mathematischen Sachverhalte. Weitere Lehrbücher zu anderen Sachgebieten werden folgen. Der Autor arbeitet an einem Buch mit dem Titel *Computational Oriented Matroids* [1], das bei Cambridge University Press erscheinen wird. Haskell 98 wird dabei im Bereich der Diskreten Geometrie eingesetzt. Funktionale Programmierung übernimmt die exakte Vermittlung von sonst eher komplexen Sachverhalten.

Eine kürzlich verfasste Arbeit mit dem Titel *Inductive Graphs and Functional Graph Algorithms* von Martin Erwig [2] zeigt Anwendungsmöglichkeiten funktionaler Sprachen in der Graphentheorie.

Das folgende Beispiel zeigt einen Haskell Code, um Primzahlen berechnen zu lassen. Beim *Sieb des Erastostenes* nehmen wir aus einer Liste ganzer Zahlen das Kopfelement, streichen die Vielfachen dieses Elementes aus der Liste und setzen dieses Siebverfahen fort. Genauer beschreibt das der folgende (einzeilige!) Haskell Code.

```
ps=sieb[2..] where sieb(p:ns)=p:sieb[n|n<-ns,n`mod`p>0]
```

Mit `take 10 ps` erhält man dann z. B. die ersten 10 Primzahlen.

Der Haskell Code für einen Sortieralgorithmus *quicksort* ist eindrucksvoll kurz.

```
qsort::[Int]->[Int]
qsort [] = []
qsort (x:xs) = (qsort smaller)++[x]++(qsort greater)
             where
             smaller = [el|el<-xs, el<x]
             greater = [el|el<-xs, el>x]
```

Wieder ist die Anschrift der mathematischen Idee nach Einhaltung der Haskell Syntax bereits der interpretierbare Code.

Die Serie einfacher Beispiele lässt sich leicht fortsetzen. Fibonacci Zahlen erhält man durch den folgenden Haskell Code.

```
fib::Int->Int
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Die ersten drei vollkommenen Zahlen erhält man leicht durch `take 3 vollkommen`, wenn die folgenden beiden Programmzeilen notiert sind.

```
faktoren n = [i|i<-[1..n-1],n`mod`i==0]
vollkommen = [n|n<-[1..],sum(faktoren n)==n]
```

Jeder Haskell Quellcode kann in ein C-Programm umgewandelt werden (z. B. durch den ghc Compiler). Dadurch lassen sich Laufzeiten kräftig verbessern. Bei vielen Problemen, insbesondere solchen, die in der Lehre eingesetzt werden, sind die bestmöglichen Laufzeiten zweitrangig. Wichtig ist oft eine lauffähige erste Programmversion bei erträglichem Programmieraufwand. Laufzeiten und Speicherplatzbedarf können später, etwa durch C-Programme oder Java-Programme verbessert werden.

Den Umgang mit LaTeX haben Sie auch mit einfachen Beispielen erlernt. Viel Freude im Umgang mit Haskell 98. Beginnen Sie z. B. mit [4].

Literatur

[1] Bokowski, J. (2005). Computational Oriented Matroids, erscheint bei Cambrige University Press.

[2] Erwig, M. (2004). Inductive Graphs and Functional Graph Algorithms. Under consideration for publication in J. Functional Programming.

[3] Hall, C. and O'Donnell, J. (2000). Discrete Mathematics Using a Computer. Springer Verlag London.

[4] Hudak, P., Peterson, J., and Fasel, J. (1997). A gentle introduction to Haskell. `http://www.haskell.org`

[5] Peyton Jones, S., ed., (2003). Haskell 98, Language and Libraries, The Revised Report, Cambridge University Press. A special issue of the Journal of Functional Programming.

Adresse des Autors
Prof. Dr. Jürgen Bokowski
Fachbereich Mathematik
Schlossgartenstraße 7
64289 Darmstadt
`bokowski@mathematik.tu-darmstadt.de`