Funktionale Programmierung mit Haskell
050071 Praktikum zur Fachdidaktik SoSe 2006
bei Martin Piskernig

Alexander Ölzant
9301547
E 190 884 423

1. August 2006

Inhaltsverzeichnis

1 Einleitung

Gerade zur schulischen Instruktion erscheint es wichtig, nicht nur die üblichen imperativen und objektorientierten Programmiersprachen zu unterrichten, sondern möglichst früh einen Einblick in funktionale Strukturen zu gewähren.

Im Gegensatz etwa zu Logo, das als funktionale Programmiersprache im Unterricht gerade mit jüngeren Kindern gerne eingesetzt wird, ist Haskell einerseits eine sauberere/reinere funktionale Sprache und wird von vielen auch als schöner gesehen, andererseits bringt es nicht den Ballast der graphischen Ausgabe inklusive Schildkröte mit, sodass es wohl als weniger „verspielt“ angesehen werden wird.

1.1 Funktionale Programmierung

Als Alternative zur objektorientierten Programmierung bietet sich die funktionale Programmierung an: bereits vor deren Erfindung wurden in funktionalen Sprachen Polymorphismus, Vererbung, Encapsulation implementiert, sodass das objektorientierte Modell aus dieser Sicht keinerlei Vorteile brachte.

Der Begriff „funktional“ist im Bereich der Programmiersprachen in unterschiedlichen Bedeutungen in Verwendung:

- funktional = reich an Funktionalität
- funktional = beinhaltet die Abarbeitung von Ausdrücken, die keinerlei Nebeneffekte (side effects) aufweisen im Gegensatz zur Abarbeitung von Befehlen.

Die FAQ der Newsgroup comp.lang.functional definiert dementsprechend funktionale Programmierung folgendermassen:

> Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.

(http://cbbrowne.com/info/functional.html)

Während also in imperativen Programmiersprachen Variablen verwendet werden, um Zwischenergebnisse zu speichern, zählt hier vor allem der Rückgabewert.

Ein weiterer positiver Aspekt der funktionalen Sprachen liegt in der Formulierung von Programmen, die der mathematischen Formelsprache ähnlicher ist als entsprechend aufbereitete Algorithmen in prozeduralen Sprachen. In

diesem Zusammenhang wird besonders Haskell gerne als *mind-expanding* bezeichnet, da sich besonders kompakter, subjektiv huebscher, lesbarer Code mit sauberen Definitionen formulieren laesst.

Auch bezüglich der Abarbeitungsreihenfolge (lazy Evaluation) bietet die funktionale Programmierung den Vorteil, dass nicht benötigte Ergebnisse gar nicht erst berechnet werden.

Die Auswertungsreihenfolge ist einerlei, und wenn es eine terminierende Auswertungsreihenfolge gibt, so terminiert auch die normal order evaluation (Theorem von Church/Rosser, Konfluenz-, Diamanteigenschaft). Zirkuläre Definitionen sind möglich, wenn auch nicht immer leicht zu formulieren.

Durch *Tail Recursion* kann Rekursion erst effizient implementiert werden: indem der Stack Frame der aufrufenden Funktion weiter verwendet wird, anstatt einen neuen anzulegen, koennen beliebige Rekursionstiefen ohne zusätzlichen Speicherverbrauch realisiert werden. [**?**]

1.2 Die Programmiersprache Haskell

Benannt ist die Sprache nach dem Mathematiker Haskell Curry (1900 - 1982), dessen Name von Christopher Strachey auch für die Benennung des Prinzip des *Currying* verwendet wurde. Dieses wurde eigentlich von Moses Schnfinkel und Gottlob Frege erfunden (die Begriffe schönen für currying und finkeln fuer uncurrying haben sich jedoch im Sprachgebrauch nicht wirklich durchgesetzt) bedeutet die Umsetzung einer Funktion mehrerer Variablen auf eine mit nur einer - intuitiv werden dabei die anderen festgesetzt.

Currying:

$$f : (X \times Y) \to Z$$

Uncurrying:

$$g : X \to (Y \to Z)$$

Die Sprache selbst entstand im Jahre 1987, als ein Kommitee gegründet und die Sprache standardisiert wurde, um eine saubere Implementierung einer funktionalen Sprache von Grund auf zu ermöglichen. Der aktuelle Standard *Haskell 98* stammt, wie der Name schon sagt, aus dem Jahr 1998.

Zum interaktiven Kennenlernen eignet sich am besten der Interpreter *hugs* (Haskell User's Gofer System), es gibt jedoch auch einen Compiler (*GHC*, Glasgow Haskell Compiler), der wie etwa bei Java oder inform Bytecode für die zugehörige Virtuelle Maschine produziert.

Beide sind Open Source und für die gängigsten Betriebssysteme (Unices, Mac OS, Windows, VMS, ...) verfügbar, wenngleich nicht unter der GPL.

Wie bereits bei der Einführung der funktionalen Programmiersprachen erwähnt sind dank der lazy evaluation zirkuläre Definitionen in Haskell for-

mulierbar, es gibt jedoch auch mindestens einen Dialekt mit einem anderen Paradigma: Eager Haskell implementiert speculative evaluation.

1.3 Andere Implementationen von Haskell

Seit seiner Entstehung wurde neben den oben genannten verbreiteten Versionen GHS und HUGS noch andere programmiert:

- Objektorientierte Versionen: Haskell++, O´Haskell und Mondrian
- Distributed Haskell
- Eager Haskell (speculative evaluation)
- Concurrent Clean: GUI-Integration, keine Monaden, sondern unique types
  http://www.cs.ru.nl/~clean/

2 Erste Schritte

Die Einführung in eine funktionale Sprache bietet natürlich einige interessante Möglichkeiten. Einfache Berechnungen und String-/Listenbeispiele koennen schon bald die list comprehension von Haskell demonstrieren

```
Prelude> 2*2
4
Prelude> ['h','e','l','l','o'] ++ " world"
"hello world"
Prelude> [n | n <- [0..10],  mod n 2 == 0 ]
[0,2,4,6,8,10]
Prelude> fac 5 where fac = product . enumFromTo 1
120
```

Ganz leicht kann auch die lazy evaluation gezeigt werden: aus der unendlich langen Liste ...

```
Prelude> enumFrom 1
[1,2,3,4,5,6,7,...
```

... wird bei Bedarf nur der tatsächlich benötigte Teil berechnet:

```
Prelude> take 10 (enumFrom 1)
[1,2,3,4,5,6,7,8,9,10]
```

Neben der Erläuterung der einfachen Datentypen und der wichtigsten Operatoren sollte wohl auf die Fehlermeldungen eingegangen werden.

```
Prelude> sum == 2)
ERROR - Syntax error in input (unexpected `)')
```

Abgesehen davon darf auch die Einführung in den Interpreter hugs nicht zu kurz kommen

```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:edit <filename>    edit file
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names exported by <modules>
:find <name>        edit module containing definition of name
:!command           shell escape
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
```

3 Rekursion

Das beliebteste Beispiel zur Erläuterung der Rekursion ist vielleicht die Implementation der Funktion zur Berechnung der *Faktoriellen*

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

Dies erfordert jedoch, dass das Ergebnis jedes Funktionsaufrufes mit n multipliziert wird und verhindert damit die Wiederbenutzung des Stack Frame (tail recursion). Durch eine Modifikation kann dieser Schönheitsfehler

behoben werden. In dieser Variante ist nur ein Stack Frame für die Rekursion notwendig, daher ist sie nicht weniger effizient und viel eleganter als eine iterative Variante:

```
fac :: [Integer] -> [Integer] -> Integer
fac c m  | m == 1 = c
         | otherwise = fac (c*m) (m-1)

fac> fac 1 99
933262154439441526816992388562667004907159682643816214685929638952175999932
299156089414639761565182862536979208272237582511852109168640000000000000000
000000
```

4 Lambda-Operator

Ebenfalls unumgänglich ist die Verwendung des Lambda-Operators zur Definition anonymer Funktionen.

```
Prelude> do_something 99 (\x -> x * x) where
                    do_something x fn = fn x
9801

msort = foldl (\s -> \t -> (\(u, v) -> u++[t]++v)
                           (break (>t) s)  ) []
Prelude> msort "ZyXVAbCdE1230"
"0123ACEVXZbdy"
```

5 Datentypen

Zusätzlich zu den anfangs besprochenen einfachen Datentypen wie Integer, Bool, String (eigtl. liste von Char) sind dann Tupel und der Zugriff auf die Elemente zu behandenln:

```
type Tier = (String, String, Int)

name   :: Tier -> String
linne  :: Tier -> String
nummer :: Tier -> Int

name (n,l,u) = n
linne (n,l,u) = l
nummer (n,l,u) = u
```

6 Unterrichtsplan

1. **Einführung 1**
   Erläuterung iterative, funktionale, objektorientierte Programmiersprachen
   diverse Beispiele
   Hinweis auf funktionale Eigenschaften (list comprehension, Rekursion, anonyme Funktionen ...)

2. **Einführung 2**
   Erklärung der Arbeitsumgebung
   hugs - Interpreter, Fehlermeldungen
   literale und traditionelle Scripte
   einfache funktionale Konstrukte

3. **Einführung 3**
   Elementare Datentypen
   Operatoren
   Konstanten
   Relatoren

4. **Rekursion**
   Vergleich iterativer und rekursiver Algorithmen
   mathematische Schreibweise
   Beispiele: Faktorielle, Fibonacci-Zahlen

5. **Rekursion Wiederholung**
   Tail recursion
   repetitive (schlichte), lineare, geschachtelte, baumartige Rekursion
   Effizienz (z. B. von Fibonacci-Implementierungen)

6. **Lambda-Operator**
   freie und gebundene Variablen
   Lambda-Ausdrücke, anonyme Funktionen
   Sortierfunktionen

7. **Layout-Konventionen, Notation, Muster** Abseitsregel

8. **Lambda-Operator Wiederholung**

9. **Tupel**
   festgelegte Zahl von Werten zur Modellierung von Strukturen

10. **Listen**
    beliebige Zahl von Werten gleichen Typs
    Kurznotation [5 .. 10]
    Listenkomprehension wieder einmal

    ```
    [ n*n | n <- list, n < 10 ]
    ```

11. **Polymorphie, Muster**
    Polymorphie == „Vielgestaltigkeit“
    Überladen == ad-hoc Polymorphie
    Muster: Werte, Variablen, Wild cards

12. **Typen**
    Produkttyp vs Tupel: beim Tupel kein Kunstruktor
    Summentyp
    Rekursive Typen (Bäume, ...)
    polymorphe Typen

13. **Polymorphie, Vererbung**
    Wiederverwendung!
    Typen, Funktionen

14. **Funktionale**
    Funktionen höherer Ordnung (involvieren Funktionen in Argumenten oder Resultaten)

15. **Currying, Funktionskomposition**
    mehrstellige Funktionen - Funktionspfeil statt Tupel

16. **Monaden**
    Ein-/Ausgabe - Schnittstelle zur imperativen Programmierung
    Festlegung der Reihenfolge von Operationen

17. **Module** (optional)
    Export/Import, Kollisionen

7 Übungsbeispiele

7.1 Rekursion: Fibonacci-Zahlen

Schreibe eine rekursive Funktion, welche die Fibonacci-Zahlen berechnet. Diese Zahlen $f_0, f_1, ...$ sind folgendermassen definiert: $f_0 = f_1 = 1$ und $f_{n+2} = f_n + f_{n+1}$.

Abzugeben ist eine rekursive Funktion `fibonacci`, welche eine natürliche Zahl n als Parameter nimmt und f_n als Ergebnis ausgibt.

7.2 Listen und Strings: Palindrome

Schreibe eine Funktion `isPal`, die fuer eine Eingabeliste oder einen String feststellt, ob sie ein Palindrom darstellt, also von vorne nach hinten die gleiche Folge von Elementen aufweist wie von hinten nach vorne.

7.3 Tupeln und Listen: Primfaktorenzerlegung

Schreibe eine Haskell-Funktion zur Primfaktorenzerlegung, die eine Liste der Primfaktoren fuer den Eingabeparameter ausgibt. Empfehlung: implementiere dazu eine Funktion, die überprüft, ob eine Zahl eine Primzahl ist.

7.4 Typklassen und ihre Instanzen

```
data Baum a = Empty |
              Node a (Baum a) (Baum a)
```

Schreibe Haskell-Funktionen, die die Hoehe des Baumes (also den längsten Weg von der Wurzel bis zum letzten Ast) und die Anzahl der Äste berechnen!

7.5 Currying

Schreibe eine curryfizierte, polymorphe Funktion `zauberwort a -> [b] -> [b]`, die eine Eingabeliste folgendermassen modifiziert:

- wenn `a == 1`, dann drehe die Reihenfolge der Elemente in der Ausgabeliste um
- wenn `a == 2`, dann erhoehe den Wert jedes Elements in der Ausgabeliste um 1
- wenn `a == 3`, dann ersetze A..Za..z durch Z..Az..a, drehe also das Alphabet um
- ansonsten gebe „Hokuspokus“ aus

Implementiere die Varianten 1 bis 3 dazu als einstellige Funktionen.

8 Fallstricke/Caveats

Eine Eigenschaft, die in vielen anderen Programmiersprachen so nicht vorkommt, ist die Bedeutung von Einrückungen. Während es grundsätzlich in fasst jeder Sprache zur Lesbarkeit beiträgt, zusammengehörige Codeblöcke entsprechend einzurücken, markieren Einrückungen in Haskell die Bindungsbereiche und müssen daher unbedingt eingehalten werden, was dafür die Typographie vereinfacht (keine Strichpunkte notwendig).

Die beiden Varianten von script (*traditional*, **.hs** und *literal*, **.lhs**) können ebenfalls zur Verwirrung beitragen.

Syntax und Semantik kann für AnfängerInnen funktionaler Programmierung verwirrend sein.

http://www.haskell.org/tutorial/pitfalls.html führt noch folgende Probleme an:

- Typisierung: keine implizite Umwandlung
- explizite Rekursion wegen der high order functions meist nicht notwendig

```
raise :: Num a => a -> [a] -> [a]
raise _ [] = []
raise x (y:ys) = x+y : raise x ys
```

eleganter:

```
raise x ys = map (x+) ys
```

9 Iterative Programmeirung

nicht zur Nachahmung, von http://www.willamette.edu/ fruehr/haskell/evolution.html

```
fac n = result (for init next done)
        where init = (0,1)
              next   (i,m) = (i+1, m * (i+1))
              done   (i,_) = i==n
              result (_,m) = m

for i n d = until d n i
```

10 Fazit

Im Besonderen für eine funktionale Programmiersprache hat Haskell also einiges zu bieten, in einem Semester sind zumindest die Grundzüge unterzubringen, ohne die SchülerInnen zu überbeanspruchen.

Auch ohne vorhergehende Erfahrungen etwa in der Unterstufe sollte es durchaus möglich sein, Rekursion, Currying und darüber hinausgehende Konstrukte nahezubringen, zumal wenn in einem weiterführenden Kurs die Konzepte erweitert und vertieft werden können, eine erstmalige Verwendung in einer höheren Schulstufe wird vermutlich zu interessanten Relativierungen der zu diesem Zeitpunkt möglicherweise bereits eingeübten imperativen Konstrukte aus anderen Programmiersprachen führen.

Durch die mathematische Formulierung der Programme ist eine direkte Kooperation mit dem Fach Mathematik möglich, keinesfalls aber zwingend notwendig. Eine funktionale Grundeigenschaft ist *list comprehension*, mit Listen (und sogar dem Äquivalent des mathematischen Allquantors) kann elegant nicht nur ein Iteratives Konstrukt, sondern auch direkte Rekursion vermieden werden.

Der „freundliche“ Interpreter hugs, der meist aussagekräftige Fehlermeldungen ausgibt, scheint für den Unterricht ebenfalls sehr geeignet, da für Prototyping nicht erst Compile-Zyklen eingeschoben werden müssen.

Zuvorderst (und als Mindestmass für die Bewertung) sind wohl die Beherrschung funktionaler Konstrukte zu sehen, aus denen Programme mit einfachen Algorithmen zu konstruieren sind; weitergehende Projekte werden wohl einen grösseren Antel des umfangreichen Vokabulars nutzen.

11 Links, weitere Informationen

- Eleven Reasons to use Haskell as a Mathematician
  http://sigfpe.blogspot.com/2006/01/eleven-reasons-to-use-haskell-as.html
- Haskellwiki
  http://www.haskell.org/
- Haskell Reference (zvon.org)
  http://zvon.org/other/haskell/Outputglobal/index.html
- Functional Programming in the Real World
  http://homepages.inf.ed.ac.uk/wadler/realworld/index.html
- Funktionale Programmierung - Lehrveranstaltung des Instituts für Computersprachen an der TU Wien
  http://www.complang.tuwien.ac.at/knoop/fp185161_ws0506.html

12 Bibliographie

Literatur

[Friedman and Felleisen(1996)] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer*. MIT Press, 4 edition, 1996.

[Harvey(1985)] Brian Harvey. *Computer Science Logo Style*, volume 1. MIT Press, 1985.

[Probst(2001)] Mark Probst. *Proper Tail Recursion in C*. http://www.complang.tuwien.ac.at/, 2001. URL `http://www.complang.tuwien.ac.at/~schani/diplarb.ps`.