Why Functional Programming Matters

Im folgenden eine Antwort von *John Hughes*, zu finden in:

- Computer Journal 32(2), 98-107, 1989
- Research Topics in Functional Programming. D. Turner (Hrsg.), Addison Wesley, 1990
- http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html

"...an attempt to demonstrate to the "real world" that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are."

Übliche Argumentationen 1(4)

...funktionale Programmierung hat ihren Namen daher, weil

- Programme ausschließlich aus Funktionen bestehen
  - das Hauptprogramm ist selbst eine Funktion
  - es akzeptiert die Programmeingaben als Argumente und liefert die Programmausgabe als Resultat und
  - ist definiert durch andere Funktionen, die wiederum durch weitere Funktionen definiert sind (und sich schließlich auf primitive Funktionen abstützen)

Übliche Argumentationen 2(4)

Vorteile und Charakteristika funktionaler Programmierung. Eine gängige Zusammenfassung:

Funktionale Programme sind...

- frei von Zuweisungen und Seiteneffekten
- Funktionsaufrufe haben keinen Effekt außer ihr Resultat zu berechnen
- Fkt. Programme sind somit frei von einer Hauptfehlerquelle
- die Ausführungsreihenfolge wird irrelevant, Ausdrücke können zu einem beliebigen Zeitpunkt ausgewertet werden
- Programmierer wird davon befreit, den Kontrollfluss explizit festzulegen
- Ausdrücke können stets durch ihren Wert ersetzt werden und umgekehrt, Programme sind *referentiell transparent*
- Fkt. Programme sind dadurch einfacher einer mathematischen Behandlung zugänglich (z.B. für Korrektheitsbeweise)

Übliche Argumentationen 3(4)

...der "Standard"-Katalog von Vorteilen und Charakteristika fkt. Programmierung

- ist i.w. eine "Negativ"-Charakterisierung
  - *"It says a lot about what functional programming is* **not** (*it has no assignments, no side effects, no flow of control*) *but not much about what it is."*

Übliche Argumentationen 4(4)

Keine belastbaren Argumente für echte Vorteile?

Doch. Oft angeführt z.B.:

- Funktionale Programme sind
  - um eine Größenordnung kürzer als konventionelle Programme
  - Fkt. Programmierer sind dadurch weit produktiver

Aber warum? Begründbar mit den Vorteilen des Standardkatalogs? Durch Weglassen von Features? Kaum. Nicht überzeugend.

Fazit

- Der Standardkatalog ist unbefriedigend
- Wir brauchen eine positive Charakterisierung, was
  - funktionale Programmierung und ihre Stärke und
  - was ein “gutes” funktionales Programm

  ausmachen.

Auf dem Weg zu einer positiven Charakterisierung... 1(2)

Analogie: Strukturierte vs. nicht strukturierte Programmierung

Strukturierte Programme sind

- frei von goto-Anweisungen
- Blöcke sind frei von Mehrfachein- und -ausgängen
- einfacher mathematisch zugänglich als nicht strukturierte Programme

I.w. ebenfalls eine "Negativ"-Charakterisierung...

Auf dem Weg... 2(2)

Konzeptuell wichtiger...

Strukturierte Programme sind

- anders als nicht strukturierte Programme modular entworfen
- Strukturierte Programmierung ist aus diesem Grund produktiver
  - Kleine Module sind einfacher und schneller zu schreiben und zu warten
  - Wiederverwendung wird einfacher
  - Module können unabhängig getestet werden

Beachte: Der Verzicht auf goto-Anweisungen ist für den Produktivitätsgewinn nicht wesentlich.

- Abwesenheit von gotos unterstützt "*programming in the small*"
- Modularität unterstützt "*programming in the large*"

These

- Die Mächtigkeit einer Sprache, die modulares Design unterstützt, steht und fällt mit der Mächtigkeit der Konzepte, modulare Teillösungen zu einer Gesamtlösung zu kombinieren (Stichwort: *glue*). (Bsp.: Herstellung eines Stuhls)

- Funktionale Programmierung stellt zwei neue, besonders wirksame Mechanismen ("*glues*") für diesen Zweck zur Verfügung:

  1. *Funktionen höherer Ordnung* (*Funktionale*)

  2. *lazy evaluation*

  Modularisierung und Wiederverwendung eröffnen sich dadurch auch *konzeptuell* (und nicht nur technisch (Gültigkeitsregeln, separate compilation, etc.)) neue Möglichkeiten und werden wesentlich erleichtert

- Modularisierung (kleiner, einfacher, allgemeiner) ist die Richtschnur, an der sich ein funktionaler Programmierer orientieren sollte

I Glueing Functions Together...

Syntax angelehnt an Miranda (TM):

- Listen

```
listof X ::= nil | cons X (listof X)
```

- Abkürzungen

```
[]             kurz fuer      nil
[1]            kurz fuer      cons 1 nil
[1,2,3]        kurz fuer      cons 1 (cons 2 (cons 3 nil)))
```

- Aufsummieren einer Liste

```
sum nil              = 0
sum (cons num list) = num + sum list
```

Beobachtung

```
                          +----+
sum nil                 = | 0  |
                          +----+
                                 +----+
sum (cons num list) =  num       | +  |  sum list
                                 +----+
```

...die Berechnung der Summe kann modularisiert werden, indem ein allgemeines Rekursionsmuster und einige spezielle Operationen (hier eingerahmt dargestellt) miteinander verbunden werden.

```
sum = reduce add 0
where
  add x y = x+y
```

...woraus die Definition von `reduce` nahezu abgelesen werden kann:

```
(reduce f x) nil        = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Sofortige Vorteile

Ohne weiteren Programmieraufwand erhalten wir...

- Aufmultiplizieren einer Liste

```
product = reduce multiply 1
                   where multiply x y = x*y
```

- Test, ob ein Listenelement "wahr" ist

```
anytrue = reduce or false
```

- Test, ob alle Listenelemente "wahr" sind

```
alltrue = reduce and true
```

Intuition

Der Aufruf `reduce f a` kann so verstanden werden, dass in einer Argumentliste alle Vorkommen von

- `cons` durch `f` und von
- `nil` durch `a`

ersetzt werden.

Beispiel:

```
reduce add 0:
  cons 1 (cons 2 (cons 3 nil))
            -->  add 1 (add 2 (add 3 0)) = 6

reduce multiply 1:
  cons 1 (cons 2 (cons 3 nil))
            -->  multiply 1 (multiply 2 (multiply 3 1)) = 6
```

Weitere Anwendungen 1(4)

- Beobachtung

  `reduce cons nil` kopiert eine Argumentliste

- Damit möglich: `append a b = reduce cons b a`

Beispiel:

```
append [1,2] [3,4] = reduce cons [3,4] [1,2]
                   = (reduce cons [3,4]) (cons 1 (cons 2 nil))
                   = cons 1 (cons 2 [3,4])
                       -- ersetzen von cons durch cons und
                       -- von nil durch [3,4]
                   = [1,2,3,4]
```

Weitere Anwendungen 2(4)

- Verdoppeln der Elemente in einer Liste

```
doubleall = reduce doubleandcons nil
     where doubleandcons num list = cons (2*num) list
```

- Weiterer Modularisierungsschritt

```
doubleandcons = fandcons double
     where double n = 2*n
           fandcons f el list = cons (f el) list
```

Weitere Anwendungen 3(4)

- Nach einem weiteren Modularisierungsschritt

  ```
  fandcons f = cons . f
  ```

  wobei "." die Komposition von Funktionen bezeichnet:
  `(f . g) h = f (g h)`

Veranschaulichung:

```
fandcons f el = (cons . f) el
              = cons (f el)
```

Somit wie gewünscht:

```
fandcons f el list = cons (f el) list
```

Weitere Anwendungen 4(4)

- Damit erhalten wir insgesamt:
  `doubleall = reduce (cons . double) nil`

- Ein weiterer Modularisierungsschritt führt uns auf `map`

```
doubleall = map double
     where map f = reduce (cons . f ) nil
```

Nach dieser Vorbereitung ebenfalls unmittelbar möglich:

- Aufsummieren der Elemente einer Matrix:
  `summatrix = sum . map sum`

Zwischenfazit 1

Durch Modularisierung einer einfachen Funktion (im Beispiel `sum`) als Kombination einer

- Funktion höherer Ordnung und
- einiger einfacher spezieller Funktionen als Argumente

haben wir ein Programmgerüst (`reduce`) erhalten, mit dessen Hilfe sich viele Funktionen auf Listen ohne weiteren Programmieraufwand realisieren lassen.

Verallgemeinerungen auf komplexere Datenstrukturen 1(2)

Bäume

```
treeof X ::= node X (listof (treeof X))
```

Beispiel:

```
node 1                                                    1
   (cons (node 2 nil)                                    / \
         (cons (node 3                                  2   3
                    (cons (node 4 nil) nil))                |
               nil))                                        4
```

Verallgemeinerungen ... 2(2)

Analog zu `reduce` auf Listen führen wir jetzt `redtree` auf Bäumen ein:

```
redtree f g a (node label subtrees) =
        f label (redtree' f g a subtrees)
where
   redtree' f g a (cons subtree rest) =
              g (redtree f g a subtree) (redtree' f g a rest)
   redtree' f g a nil = a
```

Anwendungen 1(3)

- Aufsummieren der Blätter in einem Baum

  `sumtree = redtree add add 0`

Veranschaulichung am Beispiel:

```
add 1
    (add (add 2 0)
         (add (add 3
                   (add (add 4 0) 0))
              0))
= 10
```

Anwendungen 2(3)

- Erzeugen einer Liste aller Marken in einem Baum

  ```
  labels = redtree cons append nil
  ```

Veranschaulichung am Beispiel:

```
cons 1
     (append (cons 2 nil)
             (append (cons 3
                            (append (cons 4 nil) nil))
                     nil))
= [1,2,3,4]
```

Anwendungen 3(3)

- Eine zur Funktion `map` analoge Funktion auf Bäumen

```
maptree f = redtree (node . f) cons nil
```

Zwischenfazit 2 1(2)

- Die Ausdruckskraft in den vorausgegangenen Beispielen ergibt sich aus der Kombination
  - einer Funktion höherer Ordnung mit
  - einer speziellen spezialisierenden Funktion
- Ist die Funktion höherer Ordnung geschrieben, lassen sich viele weitere Funktionen nahezu ohne Aufwand realisieren

Zwischenfazit 2 2(2)

- *Lesson learnt*: Wann immer ein neuer Datentyp eingeführt wird, sollte man Funktionen höherer Ordnung zu seiner Verarbeitung schreiben.

- *Benefits*: Manipulation von Elementen des Datentyps wird einfach und Wissen über den Datentyp wird "lokalisiert".

- *Look&feel*: Wann immer neue Datenstrukturen neue Kontrollstrukturen wünschenswert machen, können diese mit obiger Methodik einfach ergänzt werden (im Kern am ehesten vergleichbar dem Konzept einer konventionellen erweiterbaren Sprache)

II Glueing Programs Together

Wenn `f` und `g` Programme sind, dann ist

```
g . f
```

ein Programm. Angesetzt auf die Eingabe `input`, berechnet es die Ausgabe

```
g (f input)
```

- Mögliche konventionelle Realisierung (glue): Kommunikation über Dateien
- Mögliche Probleme
  - Temporäre Datei zu groß
  - `f` terminiert nicht

Funktionaler “Leim”

Funktionale Sprachen mit *lazy evaluation* schaffen hier elegant Abhilfe.

Sie ermöglichen (glue):

- Zerlegung eines Problems in eine
  - *Generator*- und eine
  - *Selektorkomponente*

Intuition:

- Die Generatorkomponente “läuft” nur solange, bis die Selektorkomponente “zuschlägt”.

Ein erstes Beispiel

Berechnung von Quadratwurzeln nach Newton-Raphson

Gegeben: N   Gesucht: squareRoot(N)

Iterationsformel:

```
a(n+1) = (a(n) + N/a(n)) / 2
```

Rechtfertigung: Bei Konvergenz gegen `a` gilt:

```
      a = (a + N/a) / 2
   => 2a = a + N/a
      a = N/a
      a*a = N
      a = squareRoot(N)
```

Zum Vergleich

...ein imperatives (Fortran-) Programm

```
C      N is called ZN here so that it has the right type
          X = A0
          Y = A0 + 2.*EPS
C      The value of Y does not matter so long as ABS(X-Y).GT.EPS
100       IF (ABS(X-Y).LE.EPS) GOTO 200
          Y = X
          X = (X + ZN/X) / 2.
          GOTO 100
200       CONTINUE
C      The square root of ZN is now in X
```

Die funktionale Variante 1(4)

Berechnung des nächsten Folgeglieds/der nächsten Approximation

```
next N x = (x + N/x) / 2
```

Mit `f` als Bezeichnung dieser Funktion, sind wir also an der Berechnung der folgenden Sequenz von Approximationen interessiert:

```
[a0, f a0, f(f a0), f(f(f a0)), ...]
```

Die funktionale Variante 2(4)

Diese Berechnung leistet die folgende von uns definierte Funktion, die *Generatorkomponente*:

```
repeat f a = cons a (repeat f (f a))
```

Damit leistet folgender Aufruf insgesamt das Gewünschte:

```
repeat (next N) a0
```

Die funktionale Variante 3(4)

Beachte: Der Aufruf

```
repeat (next N) a0
```

terminiert nicht!

Abhilfe: ...Berechnung von `squareroot N` bis auf eine vorgegebene Genauigkeit `eps`, `eps` > 0, die *Selektorkomponente*.

Umsetzung:

```
within eps (cons a (cons b rest))
     = b,                           if abs(a-b) <= eps
     = within eps (cons b rest), otherwise
```

Es bleibt zu tun: Komposition der Komponenten/ "Module":

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

Die funktionale Variante 4(4)

Im vorigen Beispiel

- `repeat`... Generatorkomponente:

  `[a0, f a0, f(f a0), f(f(f a0)), ...]`

  ...potentiell unendlich, kein Limit für die Länge

- `within`... Selektorkomponente:

  $\texttt{f}^{i}$ `a0` mit `abs(`$\texttt{f}^{i}$ `a0 -` $\texttt{f}^{i+1}$ `a0) <= eps`

  ...lazy evaluation stellt sicher, dass die Selektorfunktion zum Zuge kommt $\Rightarrow$ Terminierung!

Varianten

Anderes Abbruchkriterium:

- ...statt Differenz zweier benachbarter Folgenglieder `<= eps`, der Quotient zweier benachbarter Folgenglieder `<= 1+eps`

Realisierung:

```
relative eps (cons a (cons b rest))
    = b,          if abs(a-b) <= eps * abs b
    = relative eps (cons b rest), otherwise
```

Es bleibt zu tun: Neukomposition der Komponenten/ "Module":

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

Beachte: Das "Modul" zur Berechnung der Sequenz der Approximationen kann unverändert übernommen werden.

Ein zweites Beispiel

Numerische Integration

Gegeben: Eine reellwertige Funktion `f` in einem Argument; die Endpunkte `a` und `b` eines Intervalls

Gesucht: Die Fläche unter der Kurve im Intervall von `a` bis `b`

Naive Realisierung:

...unter der Annahme, dass die Funktion `f` zwischen `a` und `b` linear verläuft.

```
easyintegrate f a b = (f a + f b) * (b-a) / 2
```

...hinreichend genau allenfalls für sehr kleine Intervalle.

Verfeinerungen 1(4)

Idee

- Halbiere das Intervall, berechne die Flächen für beide Teilintervalle nach voriger Formel und addiere die Teilergebnisse
- Setze diese Strategie iteriert fort

Letzteres leistet die Funktion:

```
integrate f a b = cons (easyintegrate f a b)
                       map addpair (zip (integrate f a mid)
                                        (integrate f mid b)))
     where mid = (a+b)/2
```

Erinnerung

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```

Verfeinerungen 2(4)

- `integrate` ist korrekt, aber ineffizient (wiederholte Berechnungen von `f a`, `f b`, `f mid`

Die folgende Variante von `integrate` ist frei von diesem Makel

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb = cons ((fa+fb)*(b-a)/2)
                         (map addpair (zip (integ f a m fa fm)
                                           (integ f m b fm fb)))
            where m = (a+b)/2
                  fm = f m
```

Verfeinerungen 3(4)

Beachte: Auch der Aufruf von

`integrate f a b`

terminiert nicht!

Abhilfe: ...Berechnung von `integrate f a b` bis auf eine vorgegebene Genauigkeit `eps`, `eps` > 0.

Umsetzung:

```
Variante A:    within eps (integrate f a b)
Variante B:    relative eps (integrate f a b)
```

Verfeinerungen 4(4)

Im vorigen Beispiel

- Generatorkomponente:

  `integrate`

  ...erzeugt potentiell unendliche Liste, kein Limit für die Länge

- Selektorkomponenten:

  `within`, **`relative`**

  ...lazy evaluation stellt sicher, dass die Selektorfunktionen zum Zuge kommen $\Rightarrow$ Terminierung!

Ein drittes Beispiel

Numerische Differentiation

Gegeben: Eine reellwertige Funktion `f` in einem Argument; eine Stelle `x`

Gesucht: Die Steigung von `f` an der Stelle `x`

Naive Realisierung:

...unter der Annahme, dass die Funktion `f` zwischen `x` und `x+h` keine "Überraschungen" aufweist

```
easydiff f x h = (f(x+h) - f x) / h
```

...hinreichend genau allenfalls für sehr kleine Werte von `h`.

Verfeinerungen 1(2)

Generierung einer “besser” werdenden Folge von Approximationen

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

Selektion einer hinreichend genauen Approximation

```
within esp (differentiate h0 f x)
```

Verfeinerungen 2(2)

- Auch im Differenzierungsbeispiel wird wieder das Kompositionsmuster aus
  - Generation (per se nicht limitiert!) und
  - Selektion (gewährleistet Terminierung dank lazy evaluation!)

  deutlich

Fazit 1(3)

These

- ...Modularität ist der Schlüssel zu erfolgreicher Programmierung im Großen

Beobachtung

- ...Module alleine reichen nicht

- ...die Fähigkeit, ein Problem in modulare Teilaufgaben zu zerlegen, hängt unmittelbar davon ab, welche Möglichkeiten zur Komposition von Modulen gegeben sind (glue!)

- ...der richtige Leim (glue!) ist hierfür entscheidend!

Fazit 2(3)

Tatsache

- Funktionale Programmierung stellt zwei neue Arten von Leim zur Verfügung
  - *Funktionen höherer Ordnung*
  - *Lazy evaluation*
- Funktionen höherer Ordnung und lazy evaluation erlauben substantiell neue, aufregende modulare Zerlegungen von Problemen (indem sie die entsprechenden Kompositionsmöglichkeiten zur Verfügung stellen). Eindrucksvolle Beispiele für diese These haben wir gesehen
- Im Leim liegt der essentielle Grund für die Kürze funktionaler Programme und die Leichtigkeit, sie zu schreiben

Fazit 3(3)

Handlungsrichtschnur

- Funktionale Programmierer sollten nach angemessener Modularisierung und Verallgemeinerung streben
  - Insbesondere, wann immer ein Programmteil unschön oder zu komplex erscheint
- Funktionale Programmierer sollten davon ausgehen, dass Funktionen höherer Ordnung und lazy evaluation seine Werkzeuge dafür sind

Lazy vs. eager Evaluation

Noch einmal betrachtet...

- Im Licht der vorherigen Argumente...
  - Die Vorteile von lazy evaluation als Leim scheinen so evident, dass lazy evaluation als Konzept zu wichtig ist, um es ein Dasein als *second-class citizen* fristen zu lassen
  - Lazy evaluation ist möglicherweise der mächtigste Leim, den funktionale Programmierung zu bieten hat
  - Der Zugriff auf ein solch mächtiges Konzept sollte daher nicht leichtfertig verstellt werden.

Ebenfalls sehr lohnenswert ist...

...die Auseinandersetzung mit folgenden Arbeiten:

- Paul Hudak. *Conception, Evolution, and Application of Functional Programming Languages*. ACM Computing Surveys, Vol. 21, No. 3, 359-411, 1989.

- Phil Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

- Simon Peyton Jones. *Wearing the Hair Shirt – A Retrospective on Haskell*. Eingeladener Vortrag auf dem 30th Annual Symposium on Principles of Programming Languages (POPL'03), 2003.
  Vortragsfolien: `http://research.microsoft.com/Users/simonpj/papers/haskell-retrospective/index.html`