Mathematik mit funktionaler Programmierung

Henning Thielemann

2005-05-27

Gewöhnlicher Programmierstil

- Starte mit altem Programmtext.
- Kopiere Programmtext aus anderen Programmen zusammen.
- Schreibe einfache Funktionen neu.
- Versuche nicht, in Berechnungsabschnitte zu unterteilen.
  Packe alles in eine Schleife oder in ein Hauptprogramm.
- Globale Variablen für kürzere Parameterlisten.
- Fehlerursachen nicht auf den Grund gehen (keine Zeit).
  Fehlerhaft verarbeitete Eingaben getrennt behandeln (Flicken).
- Übersetzerwarnungen unterdrücken oder ignorieren,
  Programm läuft schließlich auch so (keine Zeit).
- Einmal geschriebenes niemals verbessern, sonst riskiert man Lauffähigkeit.

Gewöhnlicher Programmierstil: Beispiele

- „Mach ich ja nur für mich“, „Mach ich ja nur für die Übung“, „Mach ich ja nur für die Firma“, „Will ja nur was ausprobieren“, „Geb ich ja nicht weiter“
- Beispiel 1:
  Gerd findet Fehlverhalten der IGPM-Bibliothek in speziellem Fall ($n = 0$?). Statt nach Fehler zu suchen, mit `if (n==0)` spezielle Behandlung des Falles.
- Beispiel 2:
  Änderung der Übersetzer-URL in HTML-Übersetzer
  - Schlecht: Fertig übersetzte HTML-Seiten mit Suchen&Ersetzen nachbearbeiten (perl)
  - Besser: Übersetzer ändern
  - Perfekt: URL von Benutzer anpassbar

Sauberer Programmierstil

- Starte mit leerem Programm.
- Möglichst viel mit Bibliotheksfunktionen erreichen. Erleichtert das Verstehen des Programmes. Bibliotheken hoffentlich gut optimiert und getestet. Falls doch nicht $\rightarrow$ Optimierung oder Fehlerbereinigung kommt allen Benutzern zugute.
- Funktion in mehreren Programmteilen gebraucht? $\rightarrow$ verallgemeinern, auslagern
- Programm in kleine für sich verständliche Teile zerlegen.
- Programmtext nicht kopieren. Kopien müssen getrennt aktualisiert und von Fehlern bereinigt werden.
- Trenne reine Berechnungen von Ein-/Ausgabe.

Sprachen können keinen Stil erzwingen, aber unterstützen.
Wie kann sauberer Stil unterstützt werden?

- Modularisierung
- Statische Typenkontrolle

Typen: schwach vs. stark

- Schwach getypt: Möglichst große Wertemenge als Typ, behandle möglichst viele Werte gleich.
  - Shell, Rexx: Alles ist Text
  - MatLab: Alles ist komplexwertige Matrix, „1“ bezeichnet komplexwertige 1×1-Matrix.
  - C: Ganze Zahlen, Zeichen, Wahrheitswerte, Aufzählungen, Bitfelder – unterstützt aber nicht unterschieden
- Stark getypt: Sehr feine Unterteilung in Typen
  - Haskell, Pascal, Modula, Oberon, Ada: Strenge Unterscheidung zwischen Wahrheitswerten, Zeichen, Aufzählungen, ganzen Zahlen, Näherungswerten (Fließkommazahlen), Brüchen, Zahlen mit Fehlerschranken, komplexen Zahlen, Restklassen, Polynomen, Vektoren, Matrizen

Typen: dynamisch vs. statisch

Wann Typen festgelegt?

- dynamische Typisierung: zur Laufzeit – mehr Freiheit?
  Beispiele: MatLab, LISP, Python, allgemein Skriptsprachen, objektorientierte Typen auch in ansonsten statisch typisierten Sprachen
- statische Typisierung: zur Übersetzungszeit – schnellere Fehlererkennung!
  Beispiele: Haskell, OCaml, Pascal, Modula, Oberon, Eiffel, Ada, C, C++

Statische Polymorphie

- Typvariablen: Funktion kann für vielerlei Typen verwendet werden, trotzdem statische Sicherheit
  Beispiel: `tail :: [a] -> [a]`
- automatischer Typenabgleich (Typinferenz)
  Beispiel: weil `"Wesel" :: [Char]` ist in `tail "Wesel"` der Typ `tail :: [Char] -> [Char]`
- Typklassen: Einschränkungen für Typvariablen
  Beispiel: **Num** für Zahlentypen
  `(+) :: Num a =>a -> a -> a`

Programmieransätze und Programmflussgraphen

imperativ	geradliniger Graph
funktional	Baum
funktional, verzögert	Baum mit Schleifen, gerichteter Graph
logisch	ungerichteter Graph

- Je freier Graph wählbar, desto mehr Arbeit für Optimierer, desto weniger Arbeit für Programmierer.
- Mehr Freiheit bei Gestaltung des Programmflussgraphen = Mehr Möglichkeiten bei Modularisierung und übersichtlicher Programmierung.
- Qual der Wahl: Viele Möglichkeiten erhöhen Verantwortung des Programmierers und erschweren Auffinden des Optimums

Funktionale Programmierung

- Funktionen sind alleinige Berechnungseinheiten
- keine Abfolge von Schritten (imperativ)
- nur Beschreibung von Transformationen und Datenflüssen

Funktionale Programmierung
=
Imperative Programmierung mit Funktionen nur ohne Befehlsfolgen?

Nein!

Weglassen imperativer Elemente gibt mehr Garantien

Keine Nebenwirkungen

- Referential transparency
  Funktionswert hängt allein von Argumenten ab
  Beispiel: $\sin 1$ hat überall den gleichen Wert
  Gegenbeispiel: `IO.GetInt();`
- explizite Datenflüsse $\rightarrow$ mehr Sicherheit
- problematisch: Ein-/Ausgabe, Zufallszahlen $\rightarrow$ Monade
- einfache Programmtransformationen durch Ersetzungsregeln
  Anwendungen: Optimierung, Ableitung

```
{-# RULES
   "map/dot" forall f g.
       map f . map g = map (f . g) ;
#-}
```

Programmbeweise

- Korrektheitsbeweise für imperative Programme immer auf funktionale Formulierung bezogen
- Beispielbehauptungen:
  - `map (f . g) == map f . map g`
  - `(x ++ y) ++ z == x ++ (y ++ z)`
- Beweisbeispiel: Definition

```
data [a] = a : [a]
         | []

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

Beweisbeispiel

Beweis mit Induktion

Fall `x==[]`

```
[] ++ (y ++ z)
y ++ z
([] ++ y) ++ z
```

Fall `x/=[]`, also `x==(a:as)`

```
(a:as) ++ (y ++ z)
a : (as ++ (y ++ z))
a : ((as ++ y) ++ z)   -- Induktionsschritt
((a:(as ++ y)) ++ z)
((a:as) ++ y) ++ z
```

Unveränderliche Werte

- keine Änderungsoperationen, etwa `INC(n);`
- keine (globalen) Variablen
- stattdessen Transformation: alter Wert $\rightarrow$ neuer Wert
- Übersetzer muss optimieren
- (gedachte) gleichzeitige Existenz aller Objekte
  Vergleiche: Differentialgleichungen mit unveränderlichen Funktionen statt mit veränderlichen Größen
  Wie in Mathematik $x_{i+1} = f(x_i)$ statt $x := f(x)$
- Konstanz nicht Bestandteil von Typen (wie in C++)

Funktionen höherer Ordnung

- keine Schleifen oder andere Programmflussstrukturen statt festverdrahtet `IF`, `FOR`, `WHILE`, Funktionen höherer Ordnung **`maybe`**, **`map`**, **`foldr`**
- ```
  map :: (a -> b) -> [a] -> [b]
  map _ [] = []
  map f (x:xs) = f x : map f xs
  ```
- **`map`** `(1+) [0,1,2,3,4,5]` = `[1,2,3,4,5,6]`
- **`map`** `(>=0) [-2,-1,0,1,2]` = `[False,False,True,True,True]`
- Beachte: Keine Indizes! → Prinzip: „Führe nur ein, was du brauchst.“
  ```

Funktionen mit mehreren Parametern

- theoretisch überflüssig, aber syntaktisch unterstützt

  Uncurried Form `(a, b) -> c` Symmetrie

  Curried Form `a -> b -> c` unvollständige Auswertung
- unvollständige Auswertung

```
map       :: (a -> b) -> [a] -> [b]
map f     ::             [a] -> [b]
map f xs  ::                    [b]
(+)       :: a -> a -> a
(+) 1     ::      a -> a
(+) 1 2   ::           a
(1+)      ::      a -> a
(1+2)     ::           a
```

Funktionale Notationen

- Lambda-Notation: $\lambda \quad x \mapsto x^2 + 1$
- Eta-Reduktion:

```
f  x  y  =  g  a  b  x  y
f  x     =  g  a  b  x
f        =  g  a  b
f        =  \x  ->  g  a  b  x
f        =  \x  ->  \y  ->  g  a  b  x  y
f        =  \x  y  ->  g  a  b  x  y
```

Modularisierung durch HOF: `all`

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
```

```
and :: [Bool] -> Bool
and = foldr (&&) True
```

```
(&&) :: Bool -> Bool -> Bool
(&&) False _ = False
(&&) _     y = y
```

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

Dokumentation durch Typen

- Analog: `sum = foldl (+) 0`
- Kristian sucht Funktion, welche Signal numerisch integriert
  Wie sucht man Funktion, deren Existenz man vermutet aber deren Namen man nicht kennt?
  - Naheliegende Namen probieren?
  - Struktur der Dokumentation zur Einschachtelung verwenden?
  - Kollegen fragen?
  - Selbst schreiben?

  Haskells Lösung
  - Typen verraten Parameter und Funktionswert
  - durchsuche Haskell-Modul `Data.List` nach Funktion, welche Liste in Liste umwandelt (`[?] -> [?]`)
  - finde `scanl :: (a -> b -> a) -> a -> [b] -> [a]`
  - kombiniere Lösung `scanl (+) 0`

Rechnen mit Funktionen: Strecken und Stauchen

```
shrink :: Num a =>
    a -> (a -> b) -> (a -> b)
shrink t f x = f (x * t)

dilate :: Fractional a =>
    a -> (a -> b) -> (a -> b)
dilate t f x = f (x / t)

translateR, translateL :: Num a =>
    a -> (a -> b) -> (a -> b)
translateR t f x = f (x - t)
translateL t f x = f (x + t)
```

Rechnen mit Funktionen: Verketten und Ableiten

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g x = f (g x)
-- alternativ:  (f . g) x = f (g x)

diff :: Fractional a =>
    a -> (a -> a) -> (a -> a)
diff eps f x =
    let epsh = eps/2
    in  (f(x+epsh) - f(x-epsh)) / eps
```

Polynomauswertung

Horner-Schema

```
horner :: Num a => [a] -> (a -> a)
horner p x = foldr (\c s -> c+x*s) 0 p
```

`horner` ist Auswertungshomomorphismus:

- verwandelt Polynom in eine Funktion
- wenn `p` Polynom, dann ist `horner p` Funktion
- Übung: Implementiere `convolve` und beweise

```
horner p0 x * horner p1 x
                    == horner (convolve p0 p1) x
```

Strikte und nicht strikte Auswertung

Strikte Auswertung

- wenn `x` undefiniert, dann `f x` undefiniert
- Undefiniertheit zum Beispiel durch Endlosschleife
- Ergebnisse immer sofort berechnet
- imperative Sprachen (fast?) immer strikt
- funktionale Sprachen manchmal strikt: LISP, OCaml

Alternative: verzögerte Auswertung (lazy evaluation)

- Ergebnis wird erst berechnet, wenn es gebraucht wird
- funktionale Sprachen: Haskell, Clean
- unendlich große Datenstrukturen
- Beispiel: endloser Ton

Neue Zerlegungsmöglichkeiten

- miteinander verwobene Berechnungen auftrennen
- Trennung von Berechnung und Ausgabe
- Beispiel Vektoraddition
  - strikt (MatLab):
    `x+y+z` rechnet erst `x+y` komplett aus, dann `(x+y)+z`
  - verzögert (Haskell):
    `zipWith` `(+)` rechnet Vektorsumme elementweise aus
    Beispiel:

```
take 10 (zipWith3 (\x y z -> x+y+z)
     (repeat 1) (repeat 2) (repeat 3))
```

Folgen, Grenzwerte

- Näherungsverfahren

  `approx :: b -> [a]`

  - Newton-Verfahren
  - Gradientenabstieg
- numerischer Grenzwert

  `lim :: [a] -> a`

  - Abbruch nach fester Anzahl
  - Abbruch falls aufeinanderfolgende Werte nur geringfügig verschieden
  - Abbruch falls nah genug an Ziel
- Konvergenzbeschleuniger

  `accelerate :: [a] -> [a]`

Reihen

- Übergang von Folge zu Reihe
  `scanl (+) 0 :: [a] -> [a]`,
  Reihenwert = Grenzwert der Partialsummenfolge
  `lim . scanl (+) 0`
- Funktionen für alle Aufgaben frei kombinierbar, weil potenziell unendliche Listen möglich

Reihengrenzwert: imperativ und strikt (Modula-3)

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

```
PROCEDURE ExpSeries (x,tol: LONGREAL;):LONGREAL =
  VAR
    sum     : LONGREAL := 0.0D0;
    summand: LONGREAL := 1.0D0;
    n       : CARDINAL := 0;
  BEGIN
    WHILE ABS(summand) > tol DO
      sum := sum + summand;
      INC(n);
      summand := summand*x / FLOAT(n, LONGREAL);
    END;
    RETURN sum;
  END ExpSeries;
```

Reihengrenzwert: funktional und verzögert (Haskell)

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

```
eval :: Double -> [Double] -> Double -> Double
eval tol ys x =
   let powers   = iterate (x*) 1
       summands = zipWith (*) ys powers
       trunc    = takeWhile (\y -> abs y > tol)
   in  sum (trunc summands)

series :: [Double]
series = scanl (/) 1 (iterate (1+) 1)

exp :: Double -> Double -> Double
exp tol = eval tol series
```

Geht das auch bei imperativen Sprachen?

- Verzögerte Auswertung bei imperativen Sprachen nahezu unmöglich, Berechnung darf nur von Parametern abhängen
- Kann MatLab Lazy? Wieso funktioniert

```
>> for k = 0:1000000000; end
```

- Vektorschreibweise 0:1000000000 ist nicht nur Zucker

```
>> for k = (5:7).^2; disp(k); end
    25
    36
    49
```

- Aber

```
>> for k = (0:1000000000)*1; end
??? Error using ==> :
Maximum variable size allowed by the prog ...
```

Explizites Euler-Verfahren für Differentialgleichung

$$\begin{aligned} u &= c_0 \cdot y + c_1 \cdot y' + y'' \\ y'' &= u - (c_0 \cdot y + c_1 \cdot y') \end{aligned}$$

```
osciODE :: Num a =>
   (a,a) -> (a,a) -> [a] -> [a]
osciODE (c0,y0) (c1,y'0) u =
   let phi ui yi y'i =
           ui - (c0 * yi + c1 * y'i)
       integrate = scanl (+)
       y   = integrate y0  y'
       y'  = integrate y'0 y''
       y'' = zipWith3 phi u y y'
   in  y
```

Wurzeln von Potenzreihen

Gegeben: Potenzreihe p
Gesucht: Potenzreihe r mit $r(t)^2 = p(t)$

$$
\begin{aligned}
p(t) &= p_0 + t \cdot p_1(t) \\
r(t) &= r_0 + t \cdot r_1(t) \\
r(t)^2 &= r_0^2 + 2 \cdot r_0 \cdot t \cdot r_1(t) + t^2 \cdot r_1(t)^2 \\
p_0 &= r_0^2 \\
r_0 &= \sqrt{p_0} \\
p_1(t) &= 2 \cdot r_0 \cdot r_1(t) + t \cdot r_1(t)^2 \\
r_1(t) &= \frac{p_1(t) - t \cdot r_1(t)^2}{2 \cdot r_0}
\end{aligned}
$$

Wurzeln von Potenzreihen

$$
\begin{aligned}
r_0 &= \sqrt{p_0} \\
r_1(t) &= \frac{p_1(t) - t \cdot r_1(t)^2}{2 \cdot r_0}
\end{aligned}
$$

```
sqRtSeries :: Floating a => [a] -> [a]
sqRtSeries (p0:p1) =
   let r0 = sqrt p0
       r1 = map (/(2*r0))
                (sub p1 (0 : mul r1 r1))
   in  r0:r1
```

Zusammenfaselung

- Funktionale Programmierung mit verzögerter Auswertung, strenger Typisierung und automatischem Typenabgleich ist supercool, wenn nicht sogar megageil.
- Alles andere ist Spielzeug.
- Ist nur was für Erwachsene.
- Wer nach der Geschwindigkeit oder dem Speicherverbrauch fragt, kriegt was vor den Latz.
- Geht endlich nach Hause.