Polynom-Klasse in Haskell

Joachim Breitner

26.6.2006

Zusammenfassung

Für mein Informatik-II-Tutorium habe ich folgende Polynom-Klasse erstellt, und veröffentliche sie hier auf Wunsch meiner Studenten. Wer das als Programmabgabe verwendet, ist selber schuld: Kein Tutor glaubt euch das (hoffe ich) :-).

Die Klasse `Polynom`

Die Typdefintion. Wir implementieren Polynome über einen beliebigen Körper (`Fractional`) nach dem Horner-Schema.

```
0 | data Fractional a ⇒ Polynom a = Nullpolynom | a `PlusXMal` (Polynom a) deriving (Eq, Ord)
```

Wir möchten eine schönere Anzeige, daher implementieren wir **Show** selbst:

```
1 | instance Fractional a ⇒ Show (Polynom a) where
2 |         show Nullpolynom                = "Nullpolynom"
3 |         show (c `PlusXMal` Nullpolynom) = show c
4 |         show (0 `PlusXMal` p)           = ""++(show p)++"·x"
5 |         show (c `PlusXMal` p)           = "("++(show p)++"·x␣+␣"++(show c)++")"
```

Man sieht den Vorteil dieser Polynomdefinition z.B. an der einfach Implementierung der Auswertungsfunktion:

```
6 | apply Nullpolynom      _ = 0
7 | apply (c `PlusXMal` p) x = c + x * apply p x
```

Zum Sparen von Tipparbeit drei Funktionen, die uns beim generieren von Polynomen helfen. Vor allem `x` ist geschickt, da man dann, nachdem man **Num** abgeleitet hat, einfach in Haskell mit `3*x*x+x*xx+3`- Polynome erzeugen kann.

```
8  | fromConst 0 = Nullpolynom
9  | fromConst c = (c `PlusXMal` Nullpolynom)
10 |
11 | xMal :: Fractional a ⇒ Polynom a → Polynom a
12 | xMal Nullpolynom = Nullpolynom
13 | xMal p           = 0 `PlusXMal` p
14 |
15 | x :: Fractional a ⇒ Polynom a
16 | x = xMal (fromConst 1)
```

Nun wollen wir **Num** ableiten:

```
17 | instance Fractional a ⇒ Num (Polynom a) where
```

Zuerst die Negation. Diese läuft einfach die Werte durch und negiert sie:

```
18 |         negate  Nullpolynom     = Nullpolynom
19 |         negate (c `PlusXMal` p) = (negate c) `PlusXMal` (negate p)
```

Die Addition der Basisfälle ist klar. Im anderen Fall vermeiden wir Nullen als Koeffizienten, indem wir den Fall $p_1 = -p_2$ abfangen und das Nullpolynom zurückgeben.

```
20 |         Nullpolynom       +  p                = p
21 |         p                 +  Nullpolynom       = p
22 |         p1@(c1 `PlusXMal` rp1) + p2@(c2 `PlusXMal` rp2)
23 |                 | p1 == (negate p2)  = Nullpolynom
24 |                 | otherwise          = (c1 + c2) `PlusXMal` (rp1 + rp2)
```

Die Multiplikation ist ähnlich. Neben den Trivialfällen bnötigen wir nur die Mittelstufenrechnung

$$(c_1 + x \cdot p_1) \cdot (c_2 + x \cdot p_2) = (c_1 \cdot c_2) + x(c_2 \cdot p_1 + c_1 \cdot p_2) + x^2(p_1 \cdot p_2)$$

Wieder stellen wir sicher, dass die Funktion sich selbst nur für kleinere Polynome aufruft, um so irgendwann beim Basisfall anzukommen.

```
25 |  Nullpolynom        *  p                 = Nullpolynom
26 |  p                  *  Nullpolynom      = Nullpolynom
27 |  (c1 `PlusXMal` p1) * (c2 `PlusXMal` p2)
28 |                     = fromConst (c1 * c2)
29 |                       + xMal (fromConst c2 * p1 + fromConst c1 * p2)
30 |                       + xMal (xMal (p1*p2))
```

Für **signum** und **abs** wählen wir den Zahlentheoretischen Ansatz: **signum** ist die Einheit des Rings der Polynome, mit der wir das Polynom nomieren (**abs**) können. Am Ende soll, nach Haskell-Doku-Empfelung gelten: **signum** `p` ∗**abs** `p` =`p`. Also ist das „Vorzeichen“ des Polynoms der Koeffizient der höchsten Potenz.

```
31 |  -- Koeffizient der höchsten Potenz
32 |  signum Nullpolynom      = Nullpolynom
33 |  signum (c `PlusXMal` p) | a == Nullpolynom = fromConst c
34 |                          | otherwise        = a
35 |                          where a = signum p
36 |
37 |  abs Nullpolynom = Nullpolynom
38 |  abs p = p * fromConst (recip (apply (signum p) 0))
```

Und Haskell ist nicht glücklich ohne:

```
39 |  fromInteger n = fromConst (fromInteger n)
```

Damit haben wir **Num** fertig implementiert. Was machen wir jetzt damit?

„Anwendungen“

Wir können beispielsweise Ableiten. Einfache Oberstufenmathematik liefert $(c + x \cdot p)' = p + x \cdot p'$, also:

```
40 | derive Nullpolynom      = Nullpolynom
41 | derive (c `PlusXMal` p) = p + x * derive p
```

Schön, und nun? Vielleicht das Newton-Verfahren zum finden von Nullstellen? Wir erinnerun uns:

$$x_{n+1} = x_n - \frac{p(x_n)}{p'(x_n)}$$

. Voilá:

```
42 | newton p s = head
43 |              $ dropWhile (\c → abs(apply p c) ≥ 0.0001)
44 |              $ iterate (\c → c - (apply p c)/(apply p' c)) s
45 |            where p' = derive p
```

Finito

Natürlich bin ich über Fehlerberichte, Ergänzungen, Verbesserungen, Kommentare und Lob jederzeit empfangsbereit, am besten per e-Mail an `mail@joachim-breitner.de`.