Christoph Lüth
Bernd Gersdorf
Christian Maeder
René Wagner
Dennis Walter
Diedrich Wolter
Simon Ulbricht

3. Übungsblatt

Ausgabe: 18.11.10
Abgabe: 29.11.10

7 *Polynome* — *10 Punkte*

Ein Polynom ist eine Summe von Vielfachen von Potenzen mit natürlichzahligen Exponenten einer Variablen, die meist mit x bezeichnet wird. Allgemein kann ein Polynom also dargestellt werden als

$$p(x) = \sum_{i=0}^{n} a_i x^i, \tag{1}$$

wobei die a_i die *Koeffizienten* des Polynoms sind. Diese müssen nicht zwingend reelle Zahlen sein, sondern können beliebigen anderen Ringen entstammen. Dem Informatiker dürfte insbesondere das Polynom über dem Ring $\mathcal{Z}_2 \equiv \{0, 1\}$ am Herzen liegen. Der zugehörige Datentyp `Bit` ist in der Datei `Bit.hs` bereitgestellt.

Die Vorlage `Blatt03.hs` definiert einen polymorphen Typ von Polynomen über einem beliebigen Ring-Typ a als Liste ihrer Koeffizienten:

```
type Poly a = [a]
```

Die Koeffizienten sollen dabei in aufsteigender Reihenfolge des zugehörigen Exponenten sortiert sein, mit der *Invariante* `last p /= 0`, d.h. der letzte Koeffizient jedes Polynoms `p` ist ungleich Null (sog. *Normalform*). Das rationale Polynom

$$x^4 + \frac{1}{2}x^3 - 2x + 1 \tag{2}$$

soll also als `[1, -2, 0, 1/2, 1] :: Poly Rational` repräsentiert werden. Ähnlich wird das Polynom (über $\mathcal{Z}_2$)

$$x^5 + x^3 + x \tag{3}$$

als `[Bit0, Bit1, Bit0, Bit1, Bit0, Bit1] :: Poly Bit` dargestellt.

Implementieren Sie die Funktionen zur Addition, Subtraktion, Multiplikation[1] und zur Ermittlung des Grades von Polynomen (entspricht dem größten Exponenten dessen Koeffizient nicht Null ist bzw. -1 für das Nullpolynom):

```
add :: Num a ⇒ Poly a → Poly a → Poly a
sub :: Num a ⇒ Poly a → Poly a → Poly a
mult :: Num a ⇒ Poly a → Poly a → Poly a
deg :: Poly a → Int
```

Machen Sie sich hierfür die folgenden Gleichungen zunutze:

$$\sum_{i=0}^{n} a_i x^i + \sum_{i=0}^{m} b_i x^i = \sum_{i=0}^{max\{m,n\}} (a_i + b_i) x^i \qquad \text{– Addition} \tag{4}$$

$$(\text{mit } a_i = 0 \text{ und } b_j = 0 \text{ für } i > n \text{ und } j > m)$$

$$cx^k \cdot \sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{n} c a_i x^{i+k} \qquad \text{– Shift} \tag{5}$$

$$\sum_{i=k}^{n} a_i x^i \cdot \sum_{i=0}^{m} b_i x^i = a_k x^k \cdot \sum_{i=0}^{m} b_i x^i + \sum_{i=k+1}^{n} a_i x^i \cdot \sum_{i=0}^{m} b_i x^i \qquad \text{– Multiplikation} \tag{6}$$

(Fürchten Sie die vielen $\sum_{i=0}^{n} a_i x^i$ nicht; sie stellen lediglich Polynome dar!)

[1] Die Typspezifikation `Num a ⇒` gibt an, dass jede Instanz des polymorphen Typs `a` die üblichen arithmetischen Operationen `+, *, -` bereitstellen muss. Sie können die Angabe inhaltlich ignorieren und einfach alle Signaturen selbstdefinierter Funktionen um diese Angabe erweitern, falls nötig. Ihre Compiler-Fehlermeldung hilft Ihnen gern.

Stellen Sie sicher, dass alle ein Polynom berechnenden Operationen die Invariante des Typs erhalten. Verfassen Sie Testfälle, die nachweisen, dass ihre Implementierung die Kommutativität der Multiplikation und das Distributivgesetz erfüllen:

```
mult p q == mult q p
mult p (add q r) == add (mult p q) (mult p r)
```

8 *Polynomdivision* — *5 Punkte*

Schließlich implementieren Sie die Polynomdivision[2]:

```
pdiv :: Fractional a ⇒ Poly a → Poly a → (Poly a, Poly a)
```

`pdiv p q` liefert ein Tupel `(s, r)`, bei dem (ähnlich der Schulbuchdivision von rationalen Zahlen) `s` den Quotienten und `r` den Rest der Division darstellen. Die charakteristische Eigenschaft dieser Funktion ist, dass für alle Polynome `p` und `q` gilt:

```
let (s, r) = pdiv p q in
add (mult s q) r == p && deg r < deg q
```

Der Grad des *Restpolynoms* `r` ist also insbesondere kleiner als der des *Divisorpolynoms* `q`. Testen Sie Ihre Implementation bzgl. der vorstehenden Eigenschaft.

Beispielrechnung:

```
2x^3 - x^2 +  x - 2 : x - 1 = 2x^2 + x + 2 Rest 0
2x^3 -2x^2
----------
       x^2 +  x - 2
       x^2 -  x
     --------------
            2x - 2
            2x - 2
            ------
                 0
```

9 *Fehlerdetektion* — *5 Punkte*

Ein interessanter Anwendungsfall für Polynome (vornehmlich vom Typ `Poly Bit`) ist die Detektion von Fehlern in Bitströmen, bekannt als *cyclic redundancy check*.

Die Grundidee ist einfach: Gegeben sei ein Polynom p über $\mathcal{Z}_2$ (hier zu verstehen als eine Bitfolge, also irgend ein Datenwert in Binärdarstellung), das über eine unsichere Leitung von Alice zu Bob übertragen werden soll. Sie möchten nun erreichen, dass Bob feststellen kann, ob das von Alice erhaltene Polynom möglicherweise durch böse α-Teilchen oder andere Einflüsse bei der Übertragung verfälscht wurde. Wir erinnern uns an die Eigenschaft der Polynomdivision ($pdiv\ p\ q = (s, r)$):

$$p = s \cdot q + r \tag{7}$$

bzw. durch Umformen

$$p - r = s \cdot q \tag{8}$$

$p - r$ ist also durch q *ohne Rest* teilbar! Nehmen wir nun an, dass q (vom Grad n) beiden Parteien bekannt ist. Teilt Alice nun $pdiv\ (x^n \cdot p)\ q = (s', r')$ und sendet $x^n \cdot p - r'$ (anstelle des eigentlichen Datums p) über die Leitung, so kann Bob beim Empfang eines Datums m erwarten, dass eine Polynomdivision $pdiv\ m\ q$ keinen Rest liefert. Andernfalls *muss* das Polynom verfälscht worden sein, d.h. $m \neq x^n \cdot p - r'$. Das Polynom r' wird häufig (etwas ungenau) als *Prüfsumme* bezeichnet.

Die Multiplikation mit x^n bewirkt, dass das Bitmuster von p in der Polynomdarstellung des übertragenen Polynoms (bei fehlerfreier Übertragung) erhalten bleibt (warum?). Es kann somit praktisch auch von Bob extrahiert werden, wenn er der Polynomdivision abgeschworen haben sollte.

[2] Wie so oft ist Wikipedia eine hilfreiche Anlaufstelle zur Auffrischung des (hier: mathematischen) Wissens.

Die Wahl eines geeigneten, d.h. möglichst viele Bitfehler detektierenden *Generatorpolynoms* q ist eine Wissenschaft für sich. Interessierte werden im Web fündig; wir konzentrieren uns hier auf ein einziges Polynom, das sogenannte CRC-16, welches unter anderem im Bluetooth-Standard Anwendung findet:

$$x^{16} + x^{12} + x^5 + 1 \tag{9}$$

Implementieren Sie die folgenden Funktionen:

```
crc16 :: Poly Bit
addCRC16 :: Poly Bit → Poly Bit
checkCRC16 :: Poly Bit → Bool
flipBit :: Int → Poly Bit → Poly Bit
test_CRC16 :: Int → Bool
```

Hierbei berechnet `addCRC16` für ein Polynom p wie oben beschrieben $x^{16} \cdot p - r'$, `checkCRC16` prüft ein "empfangenes" Polynom auf Übertragungsfehler und `flipBit n p` wandelt das `n`-te Bit in `p` um. `test_CRC16 n` testet schließlich, dass `checkCRC16` jeden einzelnen Bitkipper in der Binärdarstellung von `n` detektiert. Wer Interesse hat, kann noch eine Funktion implementieren, die für eine gegebene Bitfolge die maximale Anzahl an zusammenhängenden Bitfehlern (sog. *Burstfehler*) ermittelt, die durch das `crc16` detektiert werden.

Zum einfachen Testen stellt die Lösungsvorlage eine Funktion `asBits :: Integer →Poly Bit` bereit, die beliebig große Zahlen in Bitströme verwandelt, die dann als Testdaten verwendet werden können.

Folgende Eigenschaft soll von ihrer Implementierung erfüllt werden:

```
let p = asBits 23409534 in
let i = 13 in
checkCRC16 (addCRC16 (asBits p)) &&
  not (checkCRC16 (flipBit i (addCRC16 p)))
```