1 Polynomials

1.1 Numeral Systems

A numeral system consists of a finite set of digits, D, and a base, b, for which $b = |D|$, *i.e.* b is the cardinality of D. The binary system, for instance, uses the digits $D = \{0, 1\}$. The cardinality of D in this case, hence, is 2. The decimal system uses the digits $D = \{0 \dots 9\}$ and, thus, has the base $b = 10$. The hexadecimal system uses the digits $D = \{0 \dots 15\}$, often given as $D = \{0 \dots 9, a, b, c, d, e, f\}$, and, therefore, has the base $b = 16$.

Numbers in any numeral system are usually represented as strings of digits. The string

$$10101010,$$

for instance, may represent a number in the binary system. (It could be a number in decimal format, too, though.) The string

$$170,$$

by contrast, cannot be a binary number, because it contains the digit 7, which is not element of D in the binary system. It can represent a decimal (or a hexadecimal number). The string

$$aa,$$

can represent a number in the hexadecimal system (but not one of in the binary or decimal system).

We interpret such a string, *i.e.* convert it to the decimal system, by rewriting it as a formula of the form:

$$a_nb^n + a_{n-1}b^{n-1} + \dots + a_0b^0,$$

where a_i are the digits that appear in the string, b is the base and n is position of the left-most digit starting to count with 0 on the right-hand side of the string. The string 10101010 in binary notation,hence, is interpreted as

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0,$$

which can be simplified to

$$2^7 + 2^5 + 2^3 + 2,$$

which, in its turn, is

$$128 + 32 + 8 + 2 = 170.$$

The string 170 in decimal notation is interpreted as

$$10^2 + 7 \times 10 = 170.$$

Interpreting a string in the notation it is written in yields just that string. The string aa in hexadecimal notation is interpreted as

$$a \times 16 + a.$$

The digit a, however, is just 10. We, hence, get the equation

$$10 \times 16 + 10 = 160 + 10 = 170.$$

What do we get, when we relax some of the constraints defining a numeral system? Instead of using a finite set of digits, we could use a number field, F, (finite or infinite) so that any member of that field qualifies as coefficient in the formulas we used above to interpret numbers in the decimal system. We would then relax the rule that the base must be the cardinality of the field. Instead, we allow any member x of the field to serve as a base. Formulas we get from those new rules would follow the recipe:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 x^0$$

or shorter:

$$\sum_{i=0}^{n} a_i x^i$$

with $a_i, x \in F$.

Such beasts are indeed well-known. They are very prominent, in fact, and their name is *polynomial*.

The name *poly*nomial stems from the fact that polynomials may be composed of many terms; a monomial, by contrast, is a polynomial that consists of only one term. For instance,

$$5x^2$$

is a monomial. A binomial is a polynomial that consists of two terms. This is an example of a binomial:

$$x^5 + 2x.$$

There is nothing special about monomials and binomials, at least nothing that would affect their definition as polynomials. Monomials and binomials are just polynomials that happen to have only one or, respectively, two terms.

Polynomials share many properties with numbers. Like numbers, arithmetic, including addition, subtraction, multiplication and division as well as exponentiation, can be defined over polynomials. In some cases, numbers reveal their close relation to polynomials. The binomial theorem states, for instance, that a product of the form

$$(a + b)(a + b)$$

translates to a formula involving binomial coefficients:

$$a^2 + 2ab + b^2.$$

We can interpret this formula as the product of the polynomial $x + a$:

$$(x + a)(x + a),$$

which yields just another polynomial:

$$x^2 + 2ax + a^2$$

Let us replace a for the number 3 and fix $x = 10$. We get:

$$(10 + 3)(10 + 3) = 10^2 + 2 \times 3 \times 10 + 3^2 = 100 + 60 + 9 = 169, \tag{1.1}$$

which is just the result of the multiplication 13 × 13. Usually, it is harder to recognise this kind of relations numbers have with the binomial theorem (and, hence, with polynomials), because most binomial coefficients are too big to be represented by a single-digit number. Already in the product 14 × 14, the binomial coefficients are hidden. Let us look at this multiplication treated as the polynomial $(x + a)$ with $x = 10$ and $a = 4$:

$$(10 + 4)(10 + 4) = 10^2 + 2 \times 4 \times 10 + 4^2 = 100 + 2 \times 40 + 16.$$

When we look at the resulting number, we do not recognise the binomial coefficient anymore – they are *carried* away: $100 + 2 \times 40 + 16 = 100 + 80 + 16 = 196$.

Indeed, polynomials are not numbers. Those are different concepts. Another important difference is that polynomials do not establish a clear order. For any two distinct numbers, we can clearly say which of the two is the greater and which is the smaller one. We cannot decide that based on the formula of the polynomial alone. One way to decide quickly which of two numbers is the grater one is to look at the number of their digits. The one with more digits is necessarily the greater one. In any numeral system it holds that:

$$a_3b^3 + a_2b^2 + a_1b + a_0 > c_2b^2 + c_1b + c_0$$

independent of the values of the as and the cs. For polynomials, this is not true. Consider the following example:

$$x^3 + x^2 + x + 1 > 100x^2?$$

For $x = 10$, the left-hand side of the inequation is $1000 + 100 + 10 + 1 = 1111$; the right-hand side, however, is $100 \times 100 = 10000$.

In spite of such differences, we can represent polynomials very similar to how we represented numbers, namely as a list of coefficients. This is a valid implementation in Haskell:

```haskell
type Poly a = P [a]
  deriving (Show)
```

We add a safe constructor:

```
poly :: (Eq a, Num a) ⇒ [a] → Poly a
poly [] = error "not a polynomial"
poly as = P (cleanz as)
cleanz :: (Eq a, Num a) ⇒ [a] → [a]
cleanz xs = reverse $ go (reverse xs)
  where go []       = []
        go [0]      = [0]
        go (0 : xs) = go xs
        go xs       = xs
```

The constructor makes sure that the resulting polynomial has at least one coefficient and that all the coefficients are actually numbers and comparable for equality. The function *cleanz* called in the constructor removes leading zeros (which are redundant), just as we did when we defined natural numbers. But note that we reverse, first, the list of coefficients passed to *go* and, second, the result of *go*. This means that we store the coefficients from left to right in ascending order. Usually, we write polynomials out in descending order of their weight, *i.e.*

$$x^n + x^{n-1} + \cdots + x^0.$$

But, here, we store them in the order:

$$x^0 + x^1 + \cdots + x^{n-1} + x^n.$$

The following function gets the list of coefficients back:

```
coeffs :: Poly a → [a]
coeffs (P as) = as
```

Here is a function to pretty-print polynomials:

```
pretty :: (Num a, Show a, Eq a) ⇒ Poly a → String
pretty p = go (reverse $ weigh p)
  where go [] = ""
        go ((i, c) : cs) = let x | i ≡ 0       = ""
                                 | i ≡ 1       = "x"
                                 | otherwise = "x^" ++ show i
                               t | c ≡ 1       = x
                                 | otherwise = show c ++ x
                               o | null cs     = ""
                                 | otherwise = " + "
                           in if c ≡ 0 then go cs else t ++ o ++ go cs
weigh :: (Num a) ⇒ Poly a → [(Integer, a)]
weigh (P []) = []
weigh (P as) = (zip [0 ..] as)
```

The function demonstrates how we actually interpret the list of coefficients. We first *weigh* them by zipping the list of coefficients with a list of integers starting at 0. One could say: we count the coefficients. Note that we start with 0, so that the first coefficient gets the weight 0, the second gets the weight 1 and so on. That, again, reflects our descending ordering of coefficients.

The reversed weighted list is then passed to *go*, which does the actual printing. We first determine the substring describing x: if i, the weight, is 0, we do not want to write the x, since $x^0 = 1$. If $i = 1$, we just write x. Otherwise we write x^i.

Then we determine the term composed of coefficient and x. If the coefficient, c is 1, we just write x; otherwise, we concatenate c with x. Note, however, that we later consider an additional case, namely, when $c = 0$. In this case, we ignore the whole term.

We still consider the operation. If the remainder of the list is *null*, *i.e.* we are now handling the last term, o is the empty string. Otherwise, it is the plus symbol. Here is room for improvement: when the coefficient is negative, we do not really need the operation, since we then write $+ - cx$. Nicer would be to write only $-cx$.

Finally, we put everything together concatenating a string composed of term, operation and *go* applied on the remainder of the list.

Here is a list of polynomials and how they are represented with our Haksell type:

$x^2 + x + 1$	*poly* [1, 1, 1]
$5x^5 + 4x^4 + 3x^3 + 2x^2 + x$	*poly* [0, 1, 2, 3, 4, 5]
$5x^4 + 4x^3 + 3x^2 + 2x + 1$	*poly* [1, 2, 3, 4, 5]
$5x^4 + 3x^2 + 1$	*poly* [1, 0, 3, 0, 5]

An important concept related to polynomials is the *degree*. The degree is a measurement of the *size* of the polynomial. In concrete terms, it is the greatest exponent in the polynomial. For us, it is the weight of the right-most element in the polynomial or, much simpler, the length of the list of coefficients minus one – since, we start with zero! The following function computes the degree of a given polynomial:

```
degree :: Poly a → Int
degree (P as) = length as − 1
```

Note, by the way, that polynomials of degree 0, those with only one trivial term, are just constant numbers.

Finally, here is a useful function that creates random polynomials with *Natural* coefficients:

```
randomPoly :: Natural → Int → IO (Poly Natural)
randomPoly n d = do
  cs ← cleanz <$> mapM (\_ → randomCoeff n) [1 .. d]
  if length cs < d then randomPoly n d
    else return (P cs)

randomCoeff :: Natural → IO Natural
randomCoeff n = randomNatural (0, n − 1)
```

The function receives a *Natural* and an *Int*. The *Int* indicates the degree of the polynomial we want to obtain. The *Natural* is used to restrict the size of the coefficients we want to see in the polynomial. In *randomCoeff*, we use the *randomNatural* defined in the previous chapter to generate a random number between 0 and $n - 1$. You might suspect already where that will lead us: to polynomials modulo some number. But before we get there, we will study polynomial arithmetic.

1.2 Polynomial Arithmetic

We start with addition and subtraction, which, in German, are summarised by the beautiful word *strichrechnung* meaning literally "dash calculation" as opposed to *punktrechnung* or "dot calculation", which would be multiplication and division.

Polynomial *strichrechnung* is easy. Key is to realise that the structure of polynomials is already defined by *strichrechnung*: it is composed of terms each of which is a product of some number and a power of x. When we add (or subtract) two polynomials, we just sort them according to the exponents of their terms and add (or subtract) terms with equal exponents:

$$\begin{array}{ccccccccc} & ax^n & + & bx^{n-1} & + & \dots & + & c \\ + & dx^n & + & ex^{n-1} & + & \dots & + & f \\ = & (a+d)x^n & + & (b+e)x^{n-1} & + & \dots & + & c+f \end{array} \tag{1.2}$$

With our polynomial representation, it is easy to implement this kind of operation. One might think it was designed especially to support addition and subtraction. Here is a valid implementation:

```
add :: (Num a, Eq a) => Poly a -> Poly a -> Poly a
add = strich (+)

sub :: (Num a, Eq a) => Poly a -> Poly a -> Poly a
sub = strich (-)

strich :: (Num a, Eq a) => (a -> a -> a) -> Poly a -> Poly a -> Poly a
strich o (P x) (P y) = P (strichlist o x y)

strichlist :: (Num a, Eq a) => (a -> a -> a) -> [a] -> [a] -> [a]
strichlist o xs ys = cleanz (go xs ys)
  where go [] bs             = bs
        go as []             = as
        go (a : as) (b : bs) = a `o` b : go as bs
```

Here is one more function that might be useful later on; it folds *strichlist* on a list of lists of coefficients:

```
strichf :: (Num a, Eq a) => (a -> a -> a) -> [[a]] -> [a]
strichf o = foldl' (strichlist o) []
```

Punktrechnung, *i.e.* multiplication and division, are a bit more complex – because of the distribution law. Let us start with the simple case where we distribute a monomial over a polynomial:

```
mul1 :: Num a => (a -> a -> a) -> Int -> [a] -> a -> [a]
mul1 o i as a = zeros i ++ go as a
  where go [] _       = []
        go (c : cs) x = c `o` x : go cs x

zeros :: Num a => Int -> [a]
zeros i = take i $ repeat 0
```

The function *mul1* takes a single term (the monomial) and distributes it over the coefficients of a polynomial using the operation *o*. Each term in the polynomial is combined with the single term. This corresponds to the operation:

$$\begin{array}{ccccccccc} dx^m & \times & ax^n & + & bx^{n-1} & + & \dots & + & c \\ & = & adx^{m+n} & + & bdx^{n-1+m} & + & \dots & + & cdx^m \end{array} \tag{1.3}$$

The function *mul1* receives on more parameter, namely the *Int i* and uses it to generate a sequence of zeros that is put in front of the resulting coefficient list. As we will see shortly, the list of zeros reflects the weight of the single term. In fact, we do not implement the manipulation of the exponents we see in the abstract formula directly. Instead, the addition $+m$ is implicitly handled by placing m zeros at the head of the list resulting in a new polynomial of degree $m + d$ where d is the degree of the original polynomial. A simple example:

$$5x^2 \times (4x^3 + 3x^2 + 2x + 1) = 20x^5 + 15x^4 + 10x^3 + 5x^2$$

would be:

$\mathit{mul1}\ 2\ [1, 2, 3, 4]\ 5$

which is:

$\mathit{zero}\ 2 +\!\!+ (5 * [1, 2, 3, 4]) = [0, 0, 5, 10, 15, 20]$

We, hence, would add 2 zeros, since 2 is the degree of the monomial.

Now, when we multiply two polynomials, we need to map all terms in one of the polynomials on the other polynomial using *mul1*. We further need to pass the weight of the individual terms of the first polynomial as the *Int* parameter of *mul1*. What we want to do is:

$[\mathit{mul1}\ (*)\ i\ (\mathit{coeffs}\ \mathit{p1})\ p \mid (i, p) \leftarrow \mathit{zip}\ [0\,..]\ (\mathit{coeffs}\ \mathit{p2})]$.

What would we get applying this formula on the polynomials, say, $[1, 2, 3, 4]$ and $[5, 6, 7, 8]$? Let us have a look:

$[\mathit{mul1}\ (*)\ i\ ([5, 6, 7, 8])\ p \mid (i, p) \leftarrow \mathit{zip}\ [0\,..]\ [1, 2, 3, 4]]$
$[[5, 6, 7, 8], [0, 10, 12, 14, 16], [0, 0, 15, 18, 21, 24], [0, 0, 0, 20, 24, 28, 32]]$.

We see a list of four lists, one for each coefficient of $[1, 2, 3, 4]$. The first list is the result of distributing 1 over all the coefficients in $[5, 6, 7, 8]$. Since 1 is the first element, its weight is 0: no zeros are put before the resulting list. The second list results from distributing 2 over $[5, 6, 7, 8]$. Since 2 is the second element, its weight is 1: we add one zero. The same process is repeated for 3 and 4 resulting in the third and fourth result list. Since 3 is the the third element, the third resulting list gets two zeros and, since 4 is the fourth element, the fourth list gets three zeros.

How do we transform this list of lists back into a single list of coefficients? Very easy: we add them together using *strichf*:

$\mathit{strichf}\ (+)\ [[5, 6, 7, 8], [0, 10, 12, 14, 16], [0, 0, 15, 18, 21, 24], [0, 0, 0, 20, 24, 28, 32]]$

which is

$[5, 16, 34, 60, 61, 52, 32]$.

This means that

$$(4x^3+3x^2+2x+1)\times(8x^3+7x^2+6x+5) = 32x^6+52x^5+61x^4+60x^3+34x^2+16x+5. \quad (1.4)$$

Here is the whole algorithm:

```
mul :: (Show a, Num a, Eq a) ⇒ Poly a → Poly a → Poly a
mul p1 p2 | d2 > d1   = mul p2 p1
          | otherwise = P (strichf (+) ms)
  where d1 = degree p1
        d2 = degree p2
        ms = [mul1 (∗) i (coeffs p1) p ∨ (i, p) ← zip [0 . .] (coeffs p2)]
```

On top of multiplication, we can implement power. We will, of course, not implement a naïve approach based on repeated multiplication alone. Instead, we will use the *square-and-multiply* approach we have already used before for numbers. Here is the code:

```
powp :: (Show a, Num a, Eq a) ⇒ Natural → Poly a → Poly a
powp f poly = go f (P [1]) poly
  where go 0 y _ = y
        go 1 y x = mul y x
        go n y x | even n    = go (n `div` 2) y        (mul x x)
                 | otherwise = go ((n − 1) `div` 2) (mul y x)
                                                        (mul x x)
```

The function *powp* receives a natural number, that is the exponent, and a polynomial. We kick off by calling *go* with the exponent, f, a base polynomial $P\ [1]$, *i.e.* unity, and the polynomial we want to raise to the power of f. If $f = 0$, we are done and return the base polynomial. This reflects the case $x^0 = 1$. If $f = 1$, we multiply the base polynomial by the input polynomial. Otherwise, if the exponent is even, we halve it, pass the base polynomial on and square the input. Otherwise, we pass the product of the base polynomial and the input on instead of the base polynomial as it is. This implementation differs a bit from the implementation we presented before for numbers, but it implements the same algorithm.

Here is a simple example: we raise the polynomial $x + 1$ to the power of 5. In the first round, we compute

go 5 (P [1]) (P [1, 1]),

which, since 5 is odd, results in

go 2 (P [1, 1]) (P [1, 2, 1]).

This, in its turn, results in

go 1 (P [1, 1]) (P [1, 4, 6, 4, 1]).

This is the final step and results in

$\mathit{mul}\ (P\ [1,1])\ (P\ [1,4,6,4,1])$,

which is

$P\ [1,5,10,10,5,1]$,

the polynomial $x^5+5x^4+10x^3+10x^2+5x+1$. You might have noticed that our Haskell notation shows the binomial coefficients $\binom{n}{k}$ for $n=0$, $n=1$, $n=2$, $n=4$ and $n=5$. We never see $n=3$, which would be $P\ [1,3,3,1]$, because we leave the multiplication $\mathit{mul}\ (P\ [1,1])\ (P\ [1,2,1])$ out. For this specific case with exponent 5, leaving out this step is where square-and-multiply is more efficient than multiplying five times. With growing exponents, the saving quickly grows to a significant order.

Division is, as usual, a bit more complicated than multiplication. But it is not too different from number division. First, we define polynomial division as Euclidean division, that is we search the solution for the equation

$$\frac{a}{b} = q + r \tag{1.5}$$

where $r < b$ and $bq + r = a$.

The manual process is as follows: we divide the first term of a by the first term of b. The quotient goes to the result; then we multiply it by b and set a to a minus that result. Now we repeat the process until the degree of a is less than that of b.

Here is an example:

$$\frac{4x^5 - x^4 + 2x^3 + x^2 - 1}{x^2+1}.$$

We start by dividing $4x^5$ by x^2. The quotient is $4x^3$, which we add to the result. We multiply: $4x^3 \times (x^2+1) = 4x^5 + 4x^3$ and subtract the result from a:

$$\begin{array}{ccccccccccc}
 & 4x^5 & - & x^4 & + & 2x^3 & + & x^2 & - & 1 \\
- & 4x^5 & & & + & 4x^3 & & & & \\
= & & - & x^4 & - & 2x^3 & + & x^2 & - & 1
\end{array} \tag{1.6}$$

We continue with $-x^4$ and divide it by x^2, which is $-x^2$. The overall result now is $4x^3 - x^2$. We multiply $-x^2 \times (x^2+1) = -x^4 - x^2$ and subtract that from what remains from a:

$$\begin{array}{ccccccccc} & - & x^4 & - & 2x^3 & + & x^2 & - & 1 \\ - & - & x^4 & & & - & x^2 & & \\ = & & & - & 2x^3 & + & 2x^2 & - & 1 \end{array} \tag{1.7}$$

We continue with $-2x^3$, which, divided by x^2 is $-2x$. We multiply $-2x \times (x^2 + 1) = -2x^3 - 2x$ and subtract:

$$\begin{array}{ccccccccc} & - & 2x^3 & + & 2x^2 & + & & - & 1 \\ - & - & 2x^3 & & & - & 2x & & \\ = & & & & 2x^2 & + & 2x & - & 1 \end{array} \tag{1.8}$$

The result now is $4x^3 - x^2 - 2x$. We continue with $2x^2$, which, divided by x^2 is 2. We multiply $2 \times (x^2 + 1) = 2x^2 + 2$ and subtract:

$$\begin{array}{cccccc} & 2x^2 & + & 2x & - & 1 \\ - & 2x^2 & & & + & 2 \\ = & & & 2x & - & 3 \end{array} \tag{1.9}$$

The result now is $4x^3 - x^2 - 2x + 2$. We finally have $2x - 3$, which is smaller in degree than b. The result, hence, is $(4x^3 - x^2 - 2x + 2, 2x - 3)$.

Here is an implementation of division in Haskell:

```
divp ::    (Show a, Num a, Eq a, Fractional a, Ord a) ⇒
           Poly a → Poly a → (Poly a, Poly a)
divp (P as) (P bs) = let (q, r) = go [] as in (P q, P r)
  where db = degree (P bs)
        go q r | degree (P r) < db = (q, r)
               | null r ∨ r ≡ [0]   = (q, r)
               | otherwise          =
                 let t  = last r / last bs
                     d  = degree (P r) − db
                     ts = zeros d ++ [t]
                     m  = mulist ts bs
                 in go (cleanz $ strichlist (+) q ts)
                       (cleanz $ strichlist (−) r m)
mulist :: (Show a, Num a, Eq a) ⇒ [a] → [a] → [a]
mulist c1 c2 = coeffs $ mul (P c1) (P c2)
```

First note that division expects its arguments to be polynomials over a *Fractional* data type. We do not allow polynomials over integers to be used with this implementation. The reason is that we do not want to use Euclidean division on the coefficients. That could indeed be very confusing. Furthermore, polynomials are most often used with rational or real coefficients. Restricting division to integers (using Euclidean division) would, therefore, not make much sense.

Observe further that we call *go* with an empty set – that is the initial value of q, *i.e.* the final result – and *as* – that is initially the number to be divided, the number we called a above. The function *go* has two base cases: if the degree of r, the remainder and initially *as*, is less than the degree of the divisor b, we are done. The result is our current (q, r). The same is true if r is *null* or contains only the constant 0. In this case, there is no remainder: b divides a.

Otherwise, we divide the *last* of r by the *last* of b. Note that those are the term with the highest degree in each polynomial. This division is just a number division of the two coefficients. We still have to compute the new exponent, which is the exponent of *last* r minus the exponent of *last* b, *i.e.* their weight. We do this by subtracting their degrees and then inserting zeros at the head of the result *ts*. This result, *ts*, is then added to q. We further compute *ts* × *bs* and subtract the result from r. The function *mulist* we use for this purpose is just a wrapper around *mul* using lists of coefficients instead of *Poly* variables. With the resulting (q, r), we go into the next round.

Let us try this with our example from above:

$$\frac{4x^5 - x^4 + 2x^3 + x^2 - 1}{x^2 + 1}.$$

We call *divp* $(P\ [-1, 0, 1, 2, -1, 4])$ $(P\ [1, 0, 1])$ and get $(P\ [2, -2, -1, 4], P\ [-3, 2])$, which translates to the polynomials $4x^3 - x^2 - 2x + 2$ and $2x - 3$. This is the same result we obtained above with the manual procedure.

From here on, we can implement functions based on division, such as *divides*:

```
divides :: (Show a, Num a, Eq a, Ord a) ⇒
           Poly a → Poly a → Bool
divides a b = case b ‘divp‘ a of
                (_, P [0]) → True
                _          → False
```

the remainder:

```
remp :: (Show a, Num a, Eq a, Ord a) ⇒
        Poly a → Poly a → Bool
remp a b = let (_, r) = b ‘d‘ a in r
```

and, of course, the GCD:

```
gcdp :: (Show a, Num a, Eq a, Fractional a, Ord a) ⇒
        Poly a → Poly a → Poly a
gcdp a b | degree b > degree a = gcdp b a
         | zerop b     = a
         | otherwise = let (_, r) = divp a b in gcdp b r
```

We use a simple function to check whether a polynomial is zero:

```
zerop :: (Num a, Eq a) ⇒ Poly a → Bool
zerop (P [0]) = True
zerpo _       = False
```

We can demonstrate *gcdp* nicely on binomial coefficients. For instance, the GCD of the polynomials $x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$ and $x^3 + 3x^2 + 3x + 1$, thus

```
gcdp (P [1, 5, 10, 10, 5, 1]) (P [1, 3, 3, 1])
```

is $x^3 + 3x^2 + 3x + 1$.

Since polynomials consisting of binomial coefficients of n, where n is the degree of the polynomial, are always a product of polynomials composed of smaller binomial coefficients in the same way, the GCD of two polynomials consisting only of binomial coefficients, is always the smaller of the two. In other cases, that is, when the smaller does not divide the greater, this implementation of the GCD can lead to confusing results. For instance, we multiply $P\ [1,2,1]$ by another polynomial, say, $P\ [1,2,3]$. The result is $P\ [1,4,8,8,3]$. Now,

```
gcdp (P [1, 5, 10, 10, 5, 1]) (P [1, 4, 8, 8, 3])
```

does not yield the expected result $P\ [1,2,1]$. The reason is that the GCD is an operation defined on integers, but we implemented it on top of fractionals. That is often not what we want. Anyway, here, we will actually use the GCD only in finite fields. Until now, we have discussed polynomials in infinite fields. We now turn our attention to polynomial arithmetic in a finite field and, hence, to modular polynomial arithmetic.

With modular arithmetic, all coefficients in the polynomial are modulo n. That means we have to reduce those numbers. This, of course, does only make sense with integers. We first implement some helpers to reduce numbers modulo n reusing functions implemented in the previous chapter.

The first function takes an integer modulo n:

```
mmod :: Zahl → Zahl → Zahl
mmod n p | n < 0 ∧ (−n) > p = mmod (−(mmod (−n)) p) p
         | n < 0             = mmod (p + n) p
         | otherwise         = n ‘rem‘ p
```

Equipped with this function, we can easily implement multiplication:

```
modmul :: Zahl → Zahl → Zahl → Zahl
modmul p f1 f2 = (f1 ∗ f2) ‘mmod‘ p
```

For division, we reuse the *inverse* function:

```
modiv :: Zahl → Zahl → Zahl → Zahl
modiv p n d = modmul p n d′
  where d′ = M.inverse d p
```

Now, we turn to polynomials. Here is, first, a function that transforms a polynomial into one modulo n:

```
pmod :: Poly Zahl → Zahl → Poly Zahl
pmod (P cs) p = P [c ‘mmod‘ p | c ← cs]
```

In other words, we just map *mmod* on all coefficients. Let us look at some polynomials modulo a number, say, 7. The polynomial $P\ [1, 2, 3, 4]$ we already used above is just the same modulo 7. The polynomial $P\ [5, 6, 7, 8]$, however, changes:

$P\ [5, 6, 7, 8]$ ‘*pmod*‘ 7

is $P\ [5, 6, 0, 1]$ or, in other words, $8x^3 + 7x^2 + 6x + 5$ turns, modulo 7, into $x^3 + 6x + 5$.

The polynomial $x+1$ raised to the power of 5 is $x^5+5x^4+10x^3+10x^2+5x+1$. Modulo 7, this reduces to $x^5+5x^4+3x^3+3x^3+5x+1$. That is: the binomial coefficients modulo n change. For instance,

map (*choose2* 6) [0 . . 6]

is

1,6,15,20,15,6,1.

Modulo 7, we get

1,6,1,6,1,6,1.

map (*choose2* 7) [0 . . 7]

is

1,7,21,35,35,21,7,1.

Without big surprise, we see this modulo 7 drastically simplified:

1,0,0,0,0,0,0,1.

Here are addition and subtraction, which are very easy to convert to modular arithmetic:

```
addmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
addmp n p1 p2 = strich (+) p1 p2 ‘pmod‘ n
submp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
submp n p1 p2 = strich (−) p1 p2 ‘pmod‘ n
```

Multiplication:

```
mulmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
mulmp p p1 p2 | d2 > d1   = mulmp p p2 p1
              | otherwise = P [m ‘mmod‘ p | m ← strichf (+) ms]
  where ms = [mul1 o i (coeffs p1) c | (i, c) ← zip [0 . .] (coeffs p2)]
        d1 = degree p1
        d2 = degree p2
        o  = modmul p
```

We repeat the multiplication from above

mul (*P* [1, 2, 3, 4]) (*P* [5, 6, 7, 8])

which was

P [5, 16, 34, 60, 61, 52, 32]

Modulo 7, this result is

P [5, 2, 6, 4, 5, 3, 4].

The modulo multiplication

mulmp 7 (*P* [1, 2, 3, 4]) (*P* [5, 6, 0, 1])

yields the same result:

P [5, 2, 6, 4, 5, 3, 4]

Division:

```
divmp :: Zahl → Poly Zahl → Poly Zahl → (Poly Zahl, Poly Zahl)
divmp p (P as) (P bs) = let (q, r) = go [0] as in (P q, P r)
  where db = degree (P bs)
        go q r | degree (P r) < db = (q, r)
               | null r ∨ r ≡ [0]  = (q, r)
               | otherwise         =
                 let t  = modiv p (last r) (last bs)
                     d  = degree (P r) − db
                     ts = zeros d ++ [t]
                     m  = mulmlist p ts bs
                 in go [c ‘mmod‘ p | c ← cleanz $ strichlist (+) q ts]
                       [c ‘mmod‘ p | c ← cleanz $ strichlist (−) r m]
```

GCD:

```
gcdmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
gcdmp p a b | degree b > degree a = gcdmp p b a
            | zerop b = a
            | otherwise = let (_, r) = divmp p a b in gcdmp p b r
```

Let us try *gcdmp* on the variation we already tested above. We multiply the polynomial $x^2 + 2x + 1$ by $3x^2 + 2x + 1$ modulo 7:

mulmp 7 (P [1, 2, 1]) (P [1, 2, 3]).

The result is P [1, 4, 1, 1, 3].

Now, we compute the GCD with P [1, 5, 10, 10, 5, 1] modulo 7:

gcdmp 7 (P [1, 5, 3, 3, 5, 1]) (P [1, 4, 1, 1, 3]).

The result is P [1, 2, 1], as expected.

Finally, power:

```
powmp :: Zahl → Zahl → Poly Zahl → Poly Zahl
powmp p f poly = go f (P [1]) poly
  where go 0 y _ = y
        go 1 y x = mulmp p y x
        go n y x | even n    = go (n ‘div‘ 2) y       (mulmp p x x)
                 | otherwise = go ((n − 1) ‘div‘ 2) (mulmp p y x)
                                                    (mulmp p x x)
```

Here is a nice variant of Pascal's triangle generated by *map* ($\lambda x \to$ *powmp* 7 x (P [1, 1]) [1 . . 14]:

```
P [1, 1]
P [1, 2, 1]
P [1, 3, 3, 1]
P [1, 4, 6, 4, 1]
P [1, 5, 3, 3, 5, 1]
P [1, 6, 1, 6, 1, 6, 1]
P [1, 0, 0, 0, 0, 0, 0, 1]
P [1, 1, 0, 0, 0, 0, 0, 1, 1]
P [1, 2, 1, 0, 0, 0, 0, 1, 2, 1]
P [1, 3, 3, 1, 0, 0, 0, 1, 3, 3, 1]
P [1, 4, 6, 4, 1, 0, 0, 1, 4, 6, 4, 1]
P [1, 5, 3, 3, 5, 1, 0, 1, 5, 3, 3, 5, 1]
P [1, 6, 1, 6, 1, 6, 1, 1, 6, 1, 6, 1, 6, 1]
P [1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 1]
```

It is especially interesting to look at greater powers using exponents that are multiples of 7. Before we continue with modular arithmetic, which we need indeed to understand some of the deeper problems related to polynomials, we will investigate the application of polynomials using a famous device: Babbage's difference engine.

1.3 The Difference Engine

Polynomial arithmetic, as we have seen, is very similar to number arithmetic. What is the correspondent of interpreting a number in a given numeral system in the domain of polynomials? Well, that is the *application* of the polynomial to a given number. We would substitute x for a number in the Field in which we are working and just compute the formula. For instance, the polynomial

$$x^2 + x + 1$$

can be applied to, say, 2. Then we get the formula

$$2^2 + 2 + 1,$$

which is $4 + 2 + 1 = 7$.

For other values of x, it would of course generate other values. For $x = 0$, for instance, it would give $0^2+0+1 = 1$; for $x = 1$, it is $1^2+1+1 = 3$; for $x = 3$, it yields $3^2+3+1 = 13$.

How would we apply a polynomial represented by our Haskell type? We would need to go through the list of coefficients, raise x to the power of the weight of each particular coefficient, multiply it by the coefficient and, finally, add all the values together. Here is an implementation:

```haskell
apply :: Num a => Poly a -> a -> a
apply (P cs) x = sum [c * x ^ i | (i, c) <- zip [0 ..] cs]
```

Let us try with a very simple polynomial, $x + 1$:

apply $(P\ [1, 1])$ 0 gives 1.
apply $(P\ [1, 1])$ 1 gives 2.
apply $(P\ [1, 1])$ 2 gives 3.
apply $(P\ [1, 1])$ 3 gives 4.

This polynomial, apparently, just counts the integers adding one to the value to which we apply it. It implements `i++`.

On the first sight, this result appears to be boring. However, after a quick thought, there is a lesson to learn: we get to know the polynomial, when we look at the *sequence* it produces. So, let us implement a function that maps *apply* to lists of numbers:

```haskell
mapply :: Num a => Poly a -> [a] -> [a]
mapply p = map (apply p)
```

For simple polynomials, the sequences are predictable. x^2, obviously, just produces the squares; x^3 produces the cubes and so on. Sequences created by powers of the simple polynomial $x + 1$, still, are quite predictable, *e.g.*

mapply $(P\ [1,2,1])\ [0\,..\,10]$: 1,4,9,16,25,36,49,64,81,100,121
mapply $(P\ [1,3,3,1])\ [0\,..\,10]$: 1,8,27,64,125,216,343,512,729,1000,1331
mapply $(P\ [1,4,6,4,1])\ [0\,..\,10]$: 1,16,81,256,625,1296,2401,4096,6561,10000,14641
mapply $(P\ [1,5,10,10,5,1])\ [0\,..\,10]$:
1,32,243,1024,3125,7776,16807,32768,59049,100000,161051

The first line, easy to recognise, is the squares, but pushed one up, *i.e.* the application to 0 yields the value for 1^2, the application to 1 yields the value for 2^2 and so on. The second, still easy to recognise, is the cubes – again pushed up by one. The third line is the powers of four and the fourth line is the powers of five, both pushed up by one.

That is not too surprising at the end, since $P\ [1,2,1]$ is the result of squaring $P\ [1,1]$, which generates the integers pushed one up; $P\ [1,3,3,1]$ is the result of raising $P\ [1,1]$ to the third power and so on.

Things become more interesting, when we deviate from the binomial coefficients. The sequence produced by *mappy* $(P\ [1,2,3,4])\ [1\,..\,10]$, for instance, does not resemble such a simple sequence: 1, 10, 49, 142, 313, 586, 985, 1534, 2257, 3178, 4321. Even the Online Encyclopedia has nothing interesting to say about it. The same is true for *mappy* $(P\ [5,6,7,8])\ [1\,..\,10]$, which is 5, 26, 109, 302, 653, 1210, 2021, 3134, 4597, 6458, 8765.

This raises another interesting question: given a sequence, is there a method by which we can we recognise the polynomial that created it? Yes, there is. In fact, there are. There was even a machine that helped in guessing polynomials from sequences. It was built in the early 19^{th} century by Charles Babbage (1791 – 1871), an English polymath, mathematician, philosopher, economist and inventor.

Babbage stands in the tradition of designers and constructors of early computing machinery; predecessors of his in this tradition were, for instance, Blaise Pascal (1623 – 1662) and Gottfried Wilhelm Leibniz (1646 – 1716). Babbage designed two series of machines, first, the difference engines and, later, the analytical engines.

The analytical engine, unfortunately, was not built in his lifetime. The final collapse of the project came in 1878, after Babbage's death in 1871, due to lack of finance. The analytical engine would have been a universal (Turing-complete) computer very similar to our computers today, but not working on electricity, but on steam. It would have been programmed by punch cards that, in Babbage's time, were used for controlling looms. Programs would have resembled modern assembly languages allowing control structures like selection and iteration. In the context of a description of the analytical engine, Ada Lovelace (1815 – 1852), a friend of Babbage and daughter of Lord Byron, described how to compute Bernoulli numbers with the machine. She is, therefore, considered the first computer programmer in history.

The difference engine, at which we will look here, is much simpler. It was designed to analyse polynomials and what it did was, according to Babbage, "computing differences". During Babbage's lifetime, a first version was built and sucessfully demonstrated. The construction of a second, much more powerful version which was financially backed by the government, failed due to disputes between Babbage and his engineers. This machine was finally built by the London Science Museum in 1991 using material and engineering techniques available in the 19^{th} century proving this way that it was actually possible for Babbage and his engineers to build such a machine.

The difference engine, as Babbage put it, computes differences, namely the differences in a sequence of numbers. It would take as input a sequence of the form

0,1,16,81,256,625,1296,2401,4096,6561,10000

and compute the differences between the single numbers:

$$\begin{array}{rcrcr} 1 & - & 0 & = & 1 \\ 16 & - & 1 & = & 15 \\ 81 & - & 16 & = & 65 \\ 256 & - & 81 & = & 175 \\ \ldots & & & & \end{array} \tag{1.10}$$

Here is a simple function that does this job for us:

$$\begin{array}{ll} \mathit{diffs} :: [\mathit{Zahl}] \rightarrow [\mathit{Zahl}] & \\ \mathit{diffs}\ [\,] & = [\,] \\ \mathit{diffs}\ [\_] & = [\,] \\ \mathit{diffs}\ (a : b : cs) & = (b - a) : \mathit{diffs}\ (b : cs) \end{array}$$

Applied on the sequence above, *diffs* yields:

1,15,65,175,369,671,1105,1695,2465,3439

What is so special about it? Perhaps, nothing. But let us repeat the process using this sequence. It yields:

14,50,110,194,302,434,590,770,974

And once again:

36,60,84,108,132,156,180,204

And one more time:

24,24,24,24,24,24,24

Suddenly, we have constant list. How often did we apply *diffs*? Four times – and, as you may have realised, the original sequence was generated by the polynomial x^4, a polynomial of degree 4. Is that coincidence?

For further investigation, we implement the complete difference machine, which takes differences, until it reaches a constant sequence.

```
dengine :: [Zahl] → [[Zahl]]
dengine cs | constant cs = [ ]
           | otherwise = ds : dengine ds
  where ds = diffs cs
        constant [ ]      = True
        constant [_]      = True
        constant (x : xs) = all (≡ x) xs
```

Note that we restrict coefficients to integers. This is just for clearness. Polynomials are usually defined over a field, such as the rational or the real numbers!

To confirm our suspicion that the difference engine creates n difference sequences for a polynomial of degree n, we apply the engine on x, x^2, x^3, x^4 and x^5 and count the sequences it creates:

$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,1])\ [0\,..\,32]))$: 1
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,1])\ [0\,..\,32]))$: 2
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,0,1])\ [0\,..\,32]))$: 3
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,0,0,1])\ [0\,..\,32]))$: 4
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,0,0,0,1])\ [0\,..\,32]))$: 5

The engine already has a purpose: it tells us the degree of the polynomial that generates a given sequence. It can do much more, though. For instance, it lets us predict the next value in the sequence. To do so, we take the constant difference from the last sequence and add it to the last difference of the previous sequence; we take that result and add it to the previous sequence and so on, until we reach the first sequence. Consider the sequence and its differences from above:

0,1,16,81,256,625,1296,2401,4096,6561,10000
1,15,65,175,369,671,1105,1695,2465,3439
14,50,110,194,302,434,590,770,974
36,60,84,108,132,156,180,204
24,24,24,24,24,24,24

We start at the bottom and compute $204 + 24 = 228$. This is the next difference of the previous sequence. We compute $974 + 228 = 1202$. We go one line up and compute $3439 + 1202 = 4641$. This, finally, is the difference to the next value in the input sequence, which, hence, is $10000+4641 = 14641$ and, indeed, 11^4. Even without knowing the polynomial that actually generates the sequence, we are now able to continue this sequence. Here is a function that does that for us:

```
predict :: [[Zahl]] → [Zahl] → Maybe Zahl
predict ds xs = case go (reverse ds) of
                  0 → Nothing
                  d → Just (d + (last xs))
   where go [] = 0
         go (a : cs) = last a + go cs
```

The function takes two arguments: the first is the list of difference sequences and the second is the original sequence. We apply *go* on the reverse of the sequences (because we are working backwards). For each sequence in this list, we get the last and add it to the last of the previous until we have exhausted the list. If *go* yields 0, we assume that something went wrong. The list of sequences may have been empty in the first place. Otherwise, we add the result to the last of the original list.

Here are some more examples:

let *s* = *mapply* (*P* [0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 11
let *s* = *mapply* (*P* [0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 121
let *s* = *mapply* (*P* [0, 0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 1331
let *s* = *mapply* (*P* [0, 0, 0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 14641
let *s* = *mapply* (*P* [0, 0, 0, 0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 161051

Let us go back to the question of how to find the polynomial given the sequence that this polynomial generates. With the help of the difference engine, we already know the degree of the polynomial. Supposed, we know that the first element in the sequence was generated applying 0 to the unknown polynomial and the second one was generated applying 1, the third by applying 2 and so on, we have all information we need.

From the degree, we know the form of the polynomial. A polynomial of degree 1 has the form $a_1x + a_2$; a polynomial of degree 2 has the form $a_1x^2 + a_2x + a_3$; a polynomial of degree 3 has the form $a_1x^3 + a_2x^2 + a_3x + a_4$ and so on.

Since we know the values to which the polynomial is applied, we can easily compute the value of the x-part of the terms. They are that value raised to the power of the weight. The challenge, then, is to find the coefficient by which that value is multiplied.

The first element in the sequence is just the last coefficient, the one "without" an x, since the other terms "disappear", when we apply 0. Consider for example a polynomial of the form $x^2 + x + a$, we get $0^2 + 0 + a = c$, where c is the first value in the sequence. Thus, $a = c$.

The second element is 1 applied to the formula and, therefore, all terms equal their coefficients, since cx^n, for $x = 1$, is just c. The third element results from applying 2 to the polynomial, it hence adheres to a formula where unknown values (the coefficients) are multiplied by 2, $2^2 = 4$, $2^3 = 8$ and so on.

In other words, for a polynomial of degree n, we can devise a system of linear equations with $n + 1$ unknowns and the $n + 1$ first elements of the sequence as constant values. A polynomial of degree 2, for instance, yields the system

$$\begin{array}{rcrcrcl} & & & & a & = & a_1 \\ a & + & b & + & c & = & a_2 \\ a & + & 2b & + & 4c & = & a_3 \end{array} \tag{1.11}$$

where the constant numbers a_1, a_2 and a_3 are the first three elements of the sequence. A polynomial of degree 3 would generate the system

$$\begin{array}{rcrcrcrcl} & & & & & & a & = & a_1 \\ a & + & b & + & c & + & d & = & a_2 \\ a & + & 2b & + & 4c & + & 8d & = & a_3 \\ a & + & 3b & + & 9c & + & 27d & = & a_4 \end{array} \tag{1.12}$$

We have already learnt how to solve such systems: we can apply Gaussian elimination. The result of the elminiation is the coefficients of the generating polynomial, which are the unknowns in the linear equations. The known values (which we would call the coefficients in a linear equation) are the values obtained by computing x^i where i is the weight of the coefficient. Here is a function to extract the known values, the $x$es raised to the weight from a given sequence with a given degree:

```
genCoeff :: Zahl → Zahl → Zahl → [Zahl]
genCoeff d n x = go 0 x
  where go i x | i > d      = [x]
               | otherwise = n ↑ i : go (i + 1) x
```

Here, d is the degree of the polynomial, n is the value to which the polynomial is applied and x is the result, *i.e.* the value from the sequence. The local function *go* repeats from 0 to d, raising n, the input value, to the current weight and adding it to the resulting list. At the end, when we have reached a value greater than the degree, we add the value from the sequence as known constant yielding one line of the system of linear equations.

When we apply *genCoeff* on the the sequence generated by x^4, we would have:

genCoeff 4 0 0 resulting in $[1, 0, 0, 0, 0, 0]$
genCoeff 4 1 1 resulting in $[1, 1, 1, 1, 1, 1]$
genCoeff 4 2 16 resulting in $[1, 2, 4, 8, 16, 16]$
genCoeff 4 3 81 resulting in $[1, 3, 9, 27, 81, 81]$
genCoeff 4 4 256 resulting in $[1, 4, 16, 64, 256, 256]$

Note that the results are very regular: we see constant 1 in the first column, the natural numbers in the first column, the squares in the third, the cubes in the fourth and the powers in the fifth and sixth column. This are just the values for x^i, for $i \in \{0 \dots 4$. Since the value in the sixth column, the one we took from the sequence, equals the value in the fifth column, we can already guess that the polynomial is simply x^4. Here is another sequence, generated by a secret polynomial:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204

We compute the difference lists using *dengine* as *ds* and compute the degree of the polynomial using *length ds*. The result is 4. Now we call *genCoeff* on the first four elements of the sequence:

genCoeff 4 0 14 resulting in $[1, 0, 0, 0, 0, 14]$
genCoeff 4 1 62 resulting in $[1, 1, 1, 1, 1, 62]$
genCoeff 4 2 396 resulting in $[1, 2, 4, 8, 16, 396]$
genCoeff 4 3 1544 resulting in $[1, 3, 9, 27, 81, 1544]$
genCoeff 4 4 4322 resulting in $[1, 4, 16, 64, 256, 4322]$

We use *genCoeff* to create a matrix representing the entire system of equations:

```
findCoeffs :: [[Zahl]] → [Zahl] → L.Matrix Zahl
findCoeffs ds seq = L.M (go 0 seq)
  where d = fromIntegral (length ds)
        go _ [] = []
        go n (x : xs) | n > d      = []
                      | otherwise = genCoeff d n x : go (n + 1) xs
```

The function *findCoeffs* receives the list of difference sequences created by *dengine* and the original sequence. It computes the degree of the generating polynomial as *length ds* and, then, it goes through the first *d* elements of the sequence calling *genCoeff* with *d*, the known input value *n* and *x*, the element of the sequence. For the sequence generated by x^4, we obtain $M\ [[1, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1], [1, 2, 4, 8, 16, 16], [1, 3, 9, 27, 81, 81], [1, 4, 16, 64, 256, 256]]$, which corresponds to the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 16 \\ 1 & 3 & 9 & 27 & 81 & 81 \\ 1 & 4 & 16 & 64 & 256 & 256 \end{pmatrix}$$

For the sequence of the unknown polynomial, we obtain $M\ [[1, 0, 0, 0, 0, 14], |\ [1, 1, 1, 1, 1, 62], [1, 2, 4, 8, 16, 396], [1, 3, 9, 27, 81, 1544], [1, 4, 16, 64, 256, 4322]]$, which corresponds to the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 14 \\ 1 & 1 & 1 & 1 & 1 & 62 \\ 1 & 2 & 4 & 8 & 16 & 396 \\ 1 & 3 & 9 & 27 & 81 & 1544 \\ 1 & 4 & 16 & 64 & 256 & 4322 \end{pmatrix}$$

The next steps are simple. We create the echelon form and solve by back-substitution. The following function puts all the bits together to find the generating polynomial:

$$\begin{aligned}&\mathit{findGen} :: [[\mathit{Zahl}]] \rightarrow [\mathit{Zahl}] \rightarrow [\mathit{Quoz}]\\&\mathit{findGen}\ \mathit{ds} = \mathit{L.backsub} \circ \mathit{L.echelon} \circ \mathit{findCoeffs}\ \mathit{ds}\end{aligned}$$

Applied on the difference list and the sequence generated by x^2, *findGen* yields: $[0\ \%\ 1, 0\ \%\ 1, 0\ \%\ 1, 0\ \%\ 1, 1\ \%\ 1]$, which corresponds to the polynomial x^4. For the sequence generated by the unknown polynomial, we get: $[14\ \%\ 1, 9\ \%\ 1, 11\ \%\ 1, 16\ \%\ 1, 12\ \%\ 1]$, which corresponds to the polynomial $12x^4 + 16x^3 + 11x^2 + 9x + 14$. Let us test:

mapply $(P\ [14, 9, 11, 16, 12])\ [0\,..\,10]$ yields:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204,

which indeed is the same sequence we saw above!

Now, what about the differences generated by the difference engine? Those, too, are sequences of numbers. Are there polynomials that generate those sequences? The first difference sequence of our formerly unknown polynomial is

48,334,1148,2778,5512,9638,15444,23218,33248,45822

The next three difference sequences could be derived from this sequence – so, we can assume that this sequence is generated by a polynomial of degree 3. Let us see what *findGen* (*tail ds*) (*head ds*) yields with *ds* being the list of difference sequences of that polynomial: $[48\ \%\ 1, 118\ \%\ 1, 120\ \%\ 1, 48\ \%\ 1]$, which corresponds to the polynomial $48x^3 + 120x^2 + 118x + 48$. Let us test again:

mapply $(P\ [48, 118, 120, 48])\ [0\,..\,10]$ yields:

48,334,1148,2778,5512,9638,15444,23218,33248,45822,61228

The next difference sequence should then be generated by a polynomial of degree 2. We try with **let** $ds' = \mathit{tail}\ ds$ **in** *findGen* (*tail ds′*) (*head ds′*) and get $[286\,\%\,1, 384\,\%\,1, 144\,\%\,1]$, which corresponds to the polynomial $144x^2{+}384x{+}286$. *mapply* $(P\ [286, 384, 144])\ [0\,..\,10]$ yields:

286,814,1630,2734,4126,5806,7774,10030,12574,15406,18526

which, indeed, is the third difference sequence.

We, hence, have the three polynomials

$$\begin{array}{rcrcrcrcr}12x^4 & + & 16x^3 & + & 11x^2 & + & 9x & + & 14\\ & & 48x^3 & + & 120x^2 & + & 118x & + & 48\\ & & & & 144x^2 & + & 384x & + & 286\end{array}$$

1.4 Polynomials and Binomial Coefficients

1.5 Roots

1.6 Vieta's Formula

1.7 The Method of partial Fractions

1.8 Generationfunctionology 1

1.9 The closed Form of the Fibonacci Sequence

$$G(x) = F_0 + F_1x + F_2x^2 + F_3x^3 + \dots \tag{1.13}$$

$$G(x) = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + \dots \tag{1.14}$$

$$xG(x) = F_0x + F_1x^2 + F_2x^3 + F_3x^4 + \dots \tag{1.15}$$

$$x^2G(x) = F_0x^2 + F_1x^3 + F_2x^4 + F_3x^5 + \dots \tag{1.16}$$

$$G(x) - xG(x) - x^2G(x) = (1 - x - x^2)G(x). \tag{1.17}$$

$$\begin{array}{rlllllll}
(1-x-x^2)G(x) = & (F_0 & +F_1x & +F_2x^2 & +F_3x^3 & +\dots) & - \\
 & (& F_0x & +F_1x^2 & +F_2x^3 & +\dots) & - \\
 & (& & +F_0x^2 & +F_1x^3 & +\dots) &
\end{array}$$

$$\begin{aligned}
(1-x-x^2)G(x) = \quad & F_0 + (F_1 - F_0)x \\
& + (F_2 - F_1 - F_0)x^2 \\
& + (F_3 - F_2 - F_1)x^3 \\
& + \dots
\end{aligned}$$

$$(1 - x - x^2)G(x) = x. \tag{1.18}$$

$$G(x) = \frac{x}{1 - x - x^2}. \tag{1.19}$$

1.10 Factoring Polynomials