DTU Compute
Michael R. Hansen
25-08-2021

Mini-project: Computing with Polynomials

This exercise concerns the development of a library for computing with polynomials of the following form:

$$P(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$$

where the coefficients $a_0, \ldots, a_n$ are integers, and x is ranging over integers. It can be completed during the first 6 weeks of the semester, and is divided into 6 parts in such a way that Part i relates to some topic addressed in Week i, for $i = 1, \ldots, 6$. The aims are:

- To exercise functional programming concepts.
- To tell a coherent story, where several aspects work together in the creation of a high-quality piece of software.
- To link the story to topics from "neighbouring courses".

The themes of the various parts are:

- Part 1 focusses on *construction* of recursive list functions.
- Part 2 focusses on *functional decomposition*. In order to solve a more complicated problem, invent suitable auxiliary functions (so-called helper functions) that make the task easier.
- Part 3 focusses on analysis of your programs, where you should inspect whether your programs satisfy a so-called *invariant property*, where only certain representations of polynomials are meaningful.
- Part 4 focusses on *disjoint unions* (or *tagged values*) where you, in this exercise, should make a type for degrees of polynomials with associated operations.
- In Part 5 we will have a look at *program correctness*, that is, the implemented functions should satisfy a collection of *properties*. The strong mathematical foundation of functional programming languages makes it possible, within a course like this, to formally prove correctness of programs. We will, however, not go for mathematical correctness proofs in this course.

  Instead, we will use `FsCheck` to do *property-based testing*. In property-based testing you make programs for the correctness properties and `FsCheck` can then automatically test whether the properties hold on samples with randomly generated values. Thus, in property-based testing you construct programs (not test cases) and in this part you should apply concepts of functional programming, especially the concept of *higher-order functions*, to formulate correctness properties in a succinct manner.

- In Part 6, a *program library* `Polynomial` for polynomials is constructed in the form of an *abstract data type*, where the internal representation of polynomials is hidden from a user of the library. This library could be used in F# and in C# programs, for example. The C# program in the Appendix A produces the below presented output. This illustrates interoperability on the .Net platform:

```
p1(x) is 1 - 3x^2
p2(x) is -2x^2
p3(x) is 1 - 5x^2 + 6x^4
p4(x) is 0
Degree of p3(x) is 4
Degree of p4(x) is -infinity
Coefficients of p3(x) are 1  0  -5  0  6
p3'(x) is -10x + 24x^3
Coefficients of p1(p2(x)) are 1  0  0  0  -12
```

  In Appendix B you can see a stand-alone F# program, using the library, that produces the same output.

By completing all these parts you will exploit many aspects of functional programming. Furthermore, you will see concepts from prerequisite courses being used. For example, the concept of polynomials obviously comes from mathematics, correctness and property-based testing exploit concepts from discrete mathematics (which should be taken at least simultaneously with this course) and programming techniques, program correctness, use of invariants and property-based testing are of course relevant to computer science.

1 Part 1: Recursive list functions

We represent the polynomial $P(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$ with integer coefficients $a_0, a_1, ..., a_n$ by the list $[a_0; a_1; \ldots; a_n]$. For instance, the polynomial $2 + x^3$ is represented by the list $[2; 0; 0; 1]$. We capture this by the type declaration (actually type abbreviation):

```
type Poly = int list
```

We shall in this part implement the following operations on polynomials:

```
add:   Poly -> Poly -> Poly
mulC:  int  -> Poly -> Poly
sub:   Poly -> Poly -> Poly
mulX:  Poly         -> Poly
mul:   Poly -> Poly -> Poly
eval : int  -> Poly -> int
```

The function `add: Poly -> Poly -> Poly`

Addition of two polynomials represented by lists is performed by element-wise additions. For example, $(1+2x)+(3+4x+5x^2+6x^3) = 4+6x+5x^2+6x^3$. Notice that $(1+2x)$ is represented by the list [1;2] and $(3+4x+5x^2+6x^3)$ is represented by [3;4;5;6]. Applying `add` on these two lists should give

```
add [1;2] [3;4;5;6] = [4;6;5;6]
```

The function `mulC: int -> Poly -> Poly`

The function `mulC` should implement the multiplication of a polynomial by a constant. For example, $2 \cdot (2+x^3) = 4+2x^3$ and therefore `mulC` 2 $[2;0;0;1] = [4;0;0;2]$.

The subtraction function `sub: Poly -> Poly -> Poly`

Subtraction of two polynomials represented by lists is performed by element-wise subtractions. For example, $(1+2x)-(3+4x+5x^2+6x^3) = -2-2x-5x^2-6x^3$.

The function `mulX: Poly -> Poly`

The multiplication function `mulX` should implement the multiplication of a polynomial by x. For example, $x \cdot (2+x^3) = 2x+x^4$ and therefore `mulX` $[2;0;0;1] = [0;2;0;0;1]$.

The multiplication function `mul: Poly -> Poly -> Poly`

The following properties are useful when defining multiplication:

$$\begin{aligned} 0 \cdot Q(x) &= 0 \\ (a_0 + a_1 \cdot x + ... + a_n \cdot x^n) \cdot Q(x) & \\ &= a_0 \cdot Q(x) + x \cdot ((a_1 + a_2 \cdot x + ... + a_n \cdot x^{n-1}) \cdot Q(x)) \end{aligned}$$

For example, $(2+3x+x^3) \cdot (1+2x+3x^2) = 2+7x+12x^2+10x^3+2x^4+3x^5$.

The function `eval: int -> Poly -> int`

If $P(x)$ is a polynomial and a is an integer, then the `eval` function should compute the integer value $P(a)$.

For example, if $P(x) = 2+3x+x^3$ then $P(2) = 2+3 \cdot 2+2^3 = 16$, and, therefore, `eval` 2 $[2;3;0;1] = 16$.

2 Part 2: Functional decomposition

A simple technique when solving a complex problem is

- to partition it into smaller well-defined parts and, thereafter,
- to compose these parts to solve the original problem.

The main goal is that

a program is constructed by combining simple well-understood pieces

This is a general technique that is a natural ingredient in functional programming.

Observe that multiplication of polynomials from Part 1 is implemented by composition of operations for addition, multiplication by a constant and multiplication by x. In order words, the problem of implementing multiplication is decomposed into implementation problems for addition, etc.

When the decomposition is given, the original problem may be solved easily. But it may actually be hard to find a suitable decomposition of a complex problem.

This theme is started up in this part, where you should invent suitable auxiliary functions (helper functions) that makes programming of the following functions easy:

```
isLegal:    int list      -> bool
prune:      int list      -> Poly
toString:   Poly          -> string
derivative: Poly          -> Poly
compose:    Poly -> Poly  -> Poly
```

Following the problem formulation of Part 1, there is no *unique* representation of a polynomial. For example, $1 + 2x - 3x^2$ may be represented by the lists [1;2;-3], [1;2;-3;0] and [1;2;-3;0;0] and infinitely many others. This means that F#' built-in equality operator on lists cannot be used to compare polynomials (with the proposed representation).

We shall now prepare for a unique representation to be used in Part 3 and later parts, where we consider an integer list ns to be a *legal* representation of a polynomial only if $ns = []$ or if the last element of ns is not 0.

For example: [1;2;-3] is the only legal representation of $1 + 2x - 3x^2$ and [] is the only legal representation of the polynomial 0.

In the above types for the functions `isLegal`, ..., `compose`, we use the type `Poly` to indicate that we only want to consider values that are legal representations of polynomials. In Part 6, we shall see how a distinction between integer lists and polynomials can be enforced.

The function `isLegal: int list -> bool`

The function `isLegal` tests whether an integer lists is a legal representation of a polynomial.

The function `prune: int list -> Poly`

Any integer list can be turned into a legal representation of a polynomial by removal of 0's occurring at the end of the list. The function `prune` should do this.

The function `toString: Poly -> string`

Choose an appealing textual representation of a polynomial and declare an associated `toString` function. You may have a look at the output presented on Page 2.

The function `derivative: Poly -> Poly`

For a polynomial $P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_n \cdot x^n$, we recall that the derivative is

$$P'(x) = a_1 + 2 \cdot a_2 \cdot x + ... + n \cdot a_n \cdot x^{n-1}$$

The function `compose: Poly -> Poly -> Poly`

The composition of polynomials $P(x)$ and $Q(x)$ is defined by: $(P \circ Q)(x) = P(Q(x))$.

For example, if $P(x) = 2 + 4x^3$ and $Q(x) = 3x + 2x^2$, then

$$(P \circ Q)(x) = P(Q(x)) = 2 + 4(3x + 2x^2)^3 = 2 + 108x^3 + 216x^4 + 144x^5 + 32x^6$$

Therefore, `compose` $[2; 0; 0; 4]$ $[0; 3; 2]$ should give $[2; 0; 0; 108; 216; 144; 32]$.

Remark: It is possible to define a function computing the integral of a polynomial. This, however, requires division of numbers and for that it would be natural to base the declarations on floating point numbers rather than on integers.

However, floating point numbers (type `float`) only provide approximations of the real numbers, and these approximations will cause many extra technicalities when we consider property-based testing in Part 5. These technicalities are not particularly related to functional programming, so we stick to the integer-based types in this exercise.

Part 3: Preserving an invariant

The function `isLegal` describes a subset of the values of type `Poly`, namely the ones we consider legal (or meaningful or well-formed) representations of polynomials.

This property must be preserved by the operations on polynomials. For example, when declaring the addition function `add` $p_1\, p_2$ it can be assumed that the arguments p_1 and p_2 are legal representations of polynomials, and you are obliged to declare the function so that the result is legal as well. We say that `add` *preserves the invariant* `isLegal` when this property holds. The other functions on polynomials should also preserve this invariant.

This describes a typical situation that may pop up is different contexts. For example, in object-oriented programming, the objects of a class may be subject to some condition, a *class invariant*, and the methods of the class must preserve this invariant.

- Inspect each function declaration from Part 1 and decide whether it preserves the invariant `isLegal`.
- If a function declaration does not preserve the invariant, then revise the declaration.
- You may also consider the functions from Part 2.

Part 4: Tagged values – Degrees of polynomials

The *degree* of a polynomial $P(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$ is n when $a_n \neq 0$. The polynomial 0 is a special case and this polynomial has $-\infty$ as degree. For example:

- polynomial 0 has degree $-\infty$
- polynomial 5 has degree 0
- polynomial $2 + x^3$ has degree 3

Declare a type `Degree` having two kinds of values. The value `MinusInf` corresponds to the degree $-\infty$ and `Fin` n, where n is a (non-negative) integer, corresponds to a "normal" degree n of a polynomial.

Give a declaration for the function `degree: Poly -> Degree` computing degrees of polynomials. For example, `degree` $[] =$ `MinusInf`, `degree` $[5] =$ `Fin` 0 and `degree` $[2; 0; 0; 1] =$ `Fin` 3.

If the constructor `MinusInf` occurs before the constructor `Fin` in the declaration of `Degree`, then the built-in ordering becomes natural for degrees, where `MinusInf <=` d for any degree d, and `Fin` n_1`<= Fin` n_1 iff n_1`<=` n_2.

- Test on a few examples that the built-in operator `<=` works as intended for degrees and that the built-in `max` function does as well.
- Declare an F# function `addD: Degree -> Degree -> Degree` adding degrees. For finite degrees it boils down to adding numbers. For example, `addM (Fin 7) (fin 8)` = `Fin 15`. Furthermore, adding something to minus infinity gives minus infinity.

Part 5: Correctness - property-based testing

In this part we shall use property-based testing (in the form of `FsCheck`) to test three kinds of properties: the invariant properties considered in Part 3, well-known algebraic properties of polynomials (e.g. addition is commutative) and structure preserving properties. By completion of this part you will gain confidence in the correctness of your programs.

Preserving an invariant

Consider as an example the `add` function. It should satisfy the property:

$$\texttt{isLegal}(p_1) \wedge \texttt{isLegal}(p_2) \implies \texttt{isLegal}(\texttt{add}\ p_1\ p_2) \tag{1}$$

for all integer lists p_1 and p_2.

One could strive for mathematical proofs of properties like (1). We will, however, instead go for *property-based testing*. Conjunction and implication are easily expressed in F# and a predicate implementing (1) could be:

```
let addInvSimple p1 p2 = if isLegal p1 && isLegal p2
                         then isLegal(add p1 p2) else true
```

However, simple experiments have shown that when testing using `FsCheck` as follows

```
let _ = Check.Quick addInvSimple;;
```

a majority of the generated integer lists will violate `isLegal` and hence a majority of the generated tests will be useless. Instead we will exploit that `prune` generates legal representations of polynomials. Furthermore, experiments have shown that even after pruning there will be more that 90% different test cases from a sample. Hence, the invariant for `add` could meaningfully be tested as follows:

```
let addInv p1 p2 = isLegal(add (prune p1) (prune p2));;
let _ = Check.Quick addInv;;
```

Use property-based testing to

- check the property `isLegal`(`prune` p), that is, the result of applying `prune` will be a legal polynomial, and
- check that the functions from Part 1 preserve the invariant. You may also consider functions from Part 2.

Hint: use higher-order functions to formulate the properties in a succinct manner.

Furthermore, inject errors in some of your programs and see whether they are spotted by property-based testing.

Properties of a commutative ring

Preserving the invariant `isLegal` is one aspect of correctness for a library for polynomials. But polynomials also possess many properties that are well-known for numbers, for example, addition and multiplication of polynomials are commutative operations.

In this part you should test that your functions on polynomials satisfy the axioms (properties) of a (commutative) ring, where $+$ and $\cdot$ are binary operations, $-$ is a unary operation and `Zero` and One are polynomials.

1. For all p_1, p_2, p_3:

   $$(p_1 + p_2) + p_3 = p_1 + (p_2 + p_3)$$ $+$ (`add`) is associative

2. For all p_1, p_2:

   $$p_1 + p_2 = p_2 + p_1$$ $+$ is commutative

3. There is a polynomial Zero so that for all p:

   $$p + \text{Zero} = p = \text{Zero} + p$$ Zero is called the additive identity

4. For all p:

   $$p + (-p) = \text{Zero}$$ $-p$ is called the additive inverse of p

5. For all p_1, p_2, p_3:

   $$(p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$$ $\cdot$ (`mul`) is associative

6. For all p_1, p_2:

$$p_1 \cdot p_2 = p_2 \cdot p_1 \qquad \cdot \text{ is commutative}$$

7. There is a polynomial One so that for all p:

$$p \cdot \text{One} = p = \text{One} \cdot p \qquad \text{One is called the multiplicative identity}$$

8. For all p_1, p_2, p_3:

$$p_1 \cdot (p_2 + p_3) = p_1 \cdot p_2 + p_1 \cdot p_3$$

9. For all p_1, p_2, p_3:

$$(p_1 + p_2) \cdot p_3 = p_1 \cdot p_3 + p_2 \cdot p_3$$

where 8. and 9. express that multiplication is distributive with respect to addition.

Each of the above properties can be tested individually using the technique described above for checking that `add` preserves the invariant.

You should, however, combine this technique with functional-programming concepts to avoid repetitions of almost identical code.

For example, the associative law appears three times (Properties 1., 5. and 10. (see below)) and the commutative law twice. You may consider using techniques from Section 2.9 in the textbook to declare just one function expressing, for example, that an infix operator is associative. This function could then be applied several times.

Furthermore, certain collections of axioms express the same general property. For example,

1. 1. and 3. are the properties of a *monoid* with addition operation and Zero as identity,
2. 5. and 6. are the properties of a monoid with multiplication operation and One as identity, and
3. 10. and 11. (see below) are the properties of a monoid with composition operation and Id as identity.

You should now

- give F# declarations for additive identity Zero, multiplicative identity One and the additive inverse $-$, and
- check the properties 1. - 9. using property-based testing.

The composition function $\circ$ (i.e. `compose`) satisfies the axioms of a monoid:

10. For all p_1, p_2, p_3:

    $$(p_1 \circ p_2) \circ p_3 = p_1 \circ (p_2 \circ p_3) \qquad \circ \text{ is associative}$$

11. There is a polynomial Id so that for all p:

    $$p \circ \mathrm{Id} = p \quad \text{and} \quad \mathrm{Id} \circ p = p \qquad \text{Id is called the identity}$$

You may now

- give an F# declaration for Id and test property 11, and you may
- try to test property 10 using `Check.Quick` and experience problems with polynomials having a high degrees and possibly numerically big coefficients.

  It is possible, using FsCheck, to make your own generators where the size of numbers, length of lists, etc. can be controlled. You are not supposed to do so; but you can see in Appendix C how property 10 can be tested.

Properties of structure-preserving functions

We now address correctness by considering properties of two so-called structure-preserving functions. One is the `eval` functions that maps polynomials to integers. The other is a function `degree: Poly -> Degree` that gives the degree of a polynomial.

Properties of the `eval` function

We shall now look at and test properties of the `eval` function, where `eval` k p computes $P(k)$ when the list p is the representation of the polynomial $P(x)$. let h_k denote the function: `eval` k : `Poly` $\rightarrow$ `int`. The function h_k satisfies the properties:

12. For all k, p_1, p_2:

    $$h_k(\texttt{add}\ p_1\ p_2) = h_k(p_1) + h_k(p_2)$$

13. For all k, p_1, p_2:

    $$h_k(\texttt{mul}\ p_1\ p_2) = h_k(p_1) \cdot h_k(p_2)$$

Property 12 expresses that evaluating the sum of two polynomials `add` p_1 p_2 can be evaluated by adding the numbers achieved by individual evaluation of p_1 and p_1. Property 13 has a similar explanation.

Remark: The function h_k is in algebra called a *homomorphism*. It is a structure-preserving function, in this case, from the type `Poly` with operations `add` and `mul` to the algebra of integers with matching operations for addition and multiplication.

Validate that Properties 12 and 13 hold using property-based testing.

Degrees of polynomials

The function `degree` should satisfy the following properties:

14. For all p_1, p_2:

    $$\texttt{degree}(\texttt{add}\ p_1\ p_2) \leq \texttt{max}\ (\texttt{degree}\, p_1)\ (\texttt{degree}\, p_2)$$

15. For all p_1, p_2:

    $$\texttt{degree}(\texttt{mul}\ p_1\ p_2) = \texttt{addD}\ (\texttt{degree}\, p_1)\ (\texttt{degree}\, p_2)$$

Use property-based testing to

- validate that these two properties for `degree` hold, and to
- find a counter-example for the property

  $$\texttt{degree}(\texttt{add}\ p_1\ p_2) = \texttt{max}\ (\texttt{degree}\, p_1)\ (\texttt{degree}\, p_2)$$

Part 6: The library Polynomial

You should now implement a library for polynomials using the techniques described in Chapter 7 of the textbook.

The library should, in addition to the functions mentioned in the previous parts, include the following functions that converts between integer lists and polynomials:

```
ofList: int list -> Poly
toList: Poly     -> int list
```

and perhaps also between integer arrays and polynomials.

The internal representation of polynomials must be hidden from a user of the library, so that a user cannot violate the invariant `isLegal`.

Furthermore,

- Customize the `string` function for type `Poly`. See Section 7.7 in the textbook.
- Overload the operators `+`, `-` and `*` so that they also may operate on polynomials. See Sections 7.3 and 7.4 in the textbook.
- Hide the representation of degrees from a user of the library, customize the `string` function and overload `+`. To do so you will need techniques from Sections 7.6 and 7.7 to customize equality and ordering.

  Doing so it becomes impossible to create meaningless degrees like `Fin -2`; but you can still use the build-in functions `max` and `=` on degrees.

You may test your library from F# script files and perhaps also from programs like the ones shown in Appendices A and B.

Appendix A

```
// Polynomial.dll must be referenced
using System;
using Microsoft.FSharp.Collections; // where the type FSharpList<T> is found
namespace ConsoleApp1
{
   class Program
   {
      static void Main(string[] args)
      {   int[] n1 = new int[] {1, 0, -3 };
          FSharpList<int> n3;  // FSharp.Core must referenced

          Polynomial.Poly p1 = Polynomial.ofArray(n1);
          Polynomial.Poly p2 = Polynomial.ofArray(new int[] { 0, 0, -2});
          Polynomial.Poly p3 = p1 + p2 * p1;
          Polynomial.Poly p4 = p3 - p3;

          Polynomial.Degree d3 = Polynomial.degree(p3);
          Polynomial.Degree d4 = Polynomial.degree(p4);

          Console.WriteLine("p1(x) is " + p1);
          Console.WriteLine("p2(x) is " + p2);
          Console.WriteLine("p3(x) is " + p3);
          Console.WriteLine("p4(x) is " + p4);
          Console.WriteLine("Degree of p3(x) is " + d3);
          Console.WriteLine("Degree of p4(x) is " + d4);

          n3 = Polynomial.toList(p3);
          Console.Write("Coefficients of p3(x) are ");
          foreach (int n in n3)
             Console.Write(n + "  ");
          Console.WriteLine("");

          Console.WriteLine("p3'(x) is " + Polynomial.derivative(p3));

          n3 = Polynomial.toList(Polynomial.compose(p1, p2));
          Console.Write("Coefficients of p1(p2(x)) are ");
          foreach (int n in n3)
             Console.Write(n + "  ");
          Console.WriteLine("");
          Console.ReadKey();
     }
   }
}
```

Appendix B

```
open System

let printNumbers ns =
   let _ = List.iter (fun n -> Console.Write (string n + "   ")) ns
   Console.WriteLine ""

[<EntryPoint>]
let main argv =
   let p1 = Polynomial.ofList [1; 0; -3]
   let p2 = Polynomial.ofList [0; 0; -2]
   let p3 = p1 + p2 * p1
   let p4 = p3 - p3

   let d3 = Polynomial.degree p3
   let d4 = Polynomial.degree p4

   let _ = Console.WriteLine("p1(x) is " + string p1)
   let _ = Console.WriteLine("p2(x) is " + string p2)
   let _ = Console.WriteLine("p3(x) is " + string p3)
   let _ = Console.WriteLine("p4(x) is " + string p4)
   let _ = Console.WriteLine("Degree of p3(x) is " + string d3)
   let _ = Console.WriteLine("Degree of p4(x) is " + string d4)
   let _ = Console.Write("Coefficients of p3(x) are ")
   let _ = printNumbers(Polynomial.toList p3)

   let _ = Console.WriteLine("p3'(x) is " + string(Polynomial.derivative p3))

   let ns = Polynomial.toList(Polynomial.compose p1 p2)
   let _  = Console.Write "Coefficients of p1(p2(x)) are "
   let _  = printNumbers ns
   let _  = Console.ReadKey()
   0 // return an integer exit code
```

Appendix C

```
// Testing 10
// n is a bound on the numerical size of integers and length of lists
// m is the size of the sample
let testComp n m =
   let gen = FsCheck.Arb.generate<int list>
   let genThree = Gen.three gen
   let sample = Gen.sample n m genThree
   let prop(p1,p2,p3) =
      let pp1 = prune p1
      let pp2 = prune p2
      let pp3 = prune p2
      compose pp1 (compose pp2 pp3) = compose (compose pp1 pp2) pp3
   (List.forall prop sample,sample)

(*
> testComp 3 8;;
val it : bool * (int list * int list * int list) list =
   (true,[ ([1; -1], [0], []);
           ([], [2; 0], [0; 1; 0]);
           ([-1; 0; 0], [1; -1; -1], [-2]);
           ([-3], [], [3]);
           ([0; 1; -1], [2], [-1; 0; -1]);
           ([0; -2], [-1], [0; -1]);
           ([-3; 0], [1; -2], [1; 0; 1]);
           ([0; 2], [0], [1; 1; 0])])
*)
```